Verification of Communication Protocols in Web-Services

A thesis submitted for the degree of
Doctor of Philosophy

Anshuman Mukherjee
B.Eng.

School of Computer Science and Information Technology
College of Science, Engineering, and Health
RMIT University

August 2011

Declaration

I certify that except where due acknowledgement has been made, the work is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program; any editorial work, paid or unpaid, carried out by a third party is acknowledged; and, ethics procedures and guidelines have been followed.

Anshuman Mukherjee
School of Computer Science and Information Technology
RMIT University
August 2011

Acknowledgments

I would like to sincerely thank my supervisors, Professor Zahir Tari and Associate Professor Peter Bertok, for their support and guidance over the period of my candidature.

I am thankful to the School of Computer Science and Information Technology at RMIT University for their financial support throughout my candidature.

Sincere thanks to my fellow PhD students, in particular Vidura Gamini Abhaya and Ermyas Teshome for proof reading my papers and participating in many interesting discussions.

Finally I wish to thank my partner, Parna, and my family for their patience and support during this period.

Credits

Portions of the material in this thesis have previously appeared or are accepted to appear in the following publications:

- Anshuman Mukherjee, Zahir Tari, and Peter Bertok: A Spring Based Framework for Verification of Service Composition. In *Proceedings of the* 8^{th} *IEEE International Conference on Services Computing (SCC 2011)*, pages 258 - 265, Washington DC, USA. July 2011. IEEE Computer Society.

- Anshuman Mukherjee, Zahir Tari, and Peter Bertok: Memory efficient state-space analysis in software model-checking. In *Proceedings of the* 33^{rd} *Australasian Computer Science Conference (ACSC 2010)*, pages 23 - 32, Brisbane, Australia. January 2010. Conferences in Research and Practice in Information Technology (CRPIT).

- Anshuman Mukherjee, Zahir Tari, and Peter Bertok: Modeling BPEL composite services using clustered coloured petri-nets. In *Proceedings of the World Congress on Services II*, pages 55 - 62, Bangalore, India. September 2009. IEEE Computer Society.

- Anshuman Mukherjee, Zahir Tari, and Peter Bertok: Protocol Verification in Web Services. John Wiley & Sons, Inc., to appear in 2012

This work was supported by the Distributed Systems and Networking Discipline at RMIT University.

The thesis was written using the TexMaker editor on Windows 7 desktop, and typeset using the LaTeX 2_ε document preparation system.

Contents

Abstract **1**

1 Introduction **3**

1.1 The Problem 3
1.2 Statement of the Problem 7
1.3 Research Questions 8
1.4 Limitations of Existing Solutions 10
1.5 Contributions 12
1.6 Scope of Research 15
1.7 Thesis Structure 16

2 Background **18**

2.1 Model-checking 18
2.2 Petri Nets 20
2.3 Coloured Petri Nets 23
2.4 Business Process Execution Language (BPEL) 24
2.5 The Spring Framework 28
2.6 JAXB 2 APIs 31
2.6.1 Unmarshalling an XML document 32
2.6.2 Marshalling Java Object 32
2.7 Application of the Introduced Technologies 33

3 Memory Efficient State-Space Analysis in Software Model-Checking **34**

3.1 Motivation 34
3.2 An Overview of the Deliberated Problem & the Proposed Solution 37
3.3 Related Work 41

3.4 Proposed Models for Memory Efficient State-Space Analysis 45
3.4.1 The Sequential Model 45
A)Expanding a State in Difference Form 48
B)Decreasing the Cost of Expanding 49
C)The algorithm for Sequential Model 50
Complexity Analysis 53
3.4.2 The Tree Model 57
Algorithm for Tree Model 59
Comparison with Sequential Model 62
3.5 Experimental Results 67
3.6 Discussion 71
3.7 Summary 72

4 Time Efficient State-Space Analysis in Software Model-Checking 73
4.1 Motivation 74
4.2 An Overview of the Deliberated Problem & the Tendered Solution 76
4.3 An Overview of Hierarchical Coloured Petri-Nets 77
4.4 Related Work 84
4.5 Proposed Technique for Time Efficient State-Space Analysis 85
4.5.1 Access-Table and Parametrised Reachability Graph 87
4.5.2 Exploring a Module 90
4.5.3 Access-table & Parametrised reachability graph for a Super-Module . 96
4.5.4 Algorithms for Generating Access-Tables and parametrised reachability graphs 99
4.5.5 Additional memory cost for storing access-tables and parametrised reachability graphs 106
4.5.6 Theoretical evaluation of the reduction in delay 108
4.6 Results 111
4.6.1 Experimental Setup 112
4.6.2 Empirical Results 113
4.7 Discussion 114
4.8 Summary 115

5 Generating Hierarchical Models by Identifying Structural-Similarity 117
5.1 Motivation . . . 118
5.2 An Overview of the Deliberated Problem & the Proposed Solution . . . 120
5.3 Basics of Substitution Transition . . . 123
5.4 Related Work . . . 125
5.5 The Proposed Technique for Installing Hierarchy . . . 125
5.5.1 The Lookup Method . . . 126
Phase 1: Creating and Populating the Hash-Tables . . . 129
Phase 2: Recursively Adding Vertices . . . 133
Determining Similarity in the Example Net . . . 147
5.5.2 The Clustering Method . . . 152
5.5.3 Time Complexity of the Lookup algorithm . . . 154
5.6 Experimental Results . . . 155
5.7 Discussion . . . 161
5.8 Summary . . . 163

6 A Framework for Modeling, Simulation and Verification of a BPEL Specification 165
6.1 Motivation . . . 166
6.2 An Overview of the Deliberated Problem & the Tendered Solution . . . 169
6.3 Related Work . . . 170
6.4 The Proposed Coloured Petri-Net Semantics for BPEL . . . 172
6.4.1 Component - A . . . 173
Step 1. Instantiation . . . 174
Step 2. Dependency Injection . . . 175
6.4.2 Component - B . . . 175
Step 3. Bind . . . 175
Step 4. Marshal . . . 176
6.4.3 The Object Model for BPEL Activities . . . 178
6.4.4 The Proposed XML Templates . . . 182
Template for Global-Scope . . . 183
Template for an Activity . . . 185
Template for an Activity Synchronized by Links . . . 186
Template for Local-Scopes . . . 188

Template for Flow Activity 189
Template for Sequence Activity 191
Template for Switch Activity 192
Template for Invoke, Receive and Reply Activities 193
Template for While Activity 195
Template for Assign Activities 196
Template for Handler Activities 196
6.5 Results 198
6.5.1 Test Cases 199
6.5.2 Empirical Results 200
6.6 Discussion 203
6.7 Summary 204

7 Conclusion **206**
7.1 Results 207
7.2 Discussion 210
7.3 Future Work 213

Bibliography **216**

List of Figures

1.1 The additional states for runtime verification 4
1.2 The chronology of research. 6
1.3 A SOA based application constitutes a hierarchy of services. 7
1.4 The scope of research. 15

2.1 The process of Model-checking 20
2.2 A Petri-net model [Murata, 1989]. 22
2.3 A coloured petri-net model. Variables X and Y are of type INT. 23
2.4 Using individual exposed services for online shopping. 25
2.5 Using service composition for online shopping. 26
2.6 The modules of Spring Framework [Walls and Breidenbach, 2007] 30

3.1 S′ reached using two different sequential of events from S_0 38
3.2 The sequence of events $[e_1 e_2 \cdots e_r\rangle$ causes no net change in state 38
3.3 A Coloured Petri-Net model. Variables x and y are of type INT 39
3.4 A part of reachability graph for CPN model in Figure 3.3 39
3.5 Decision tree for boolean function $f(x_1, x_2, x_3) = \overline{x_1}.\overline{x_2}.\overline{x_3} + \overline{x_1}.x_2.\overline{x_3} + x_1.\overline{x_2}.\overline{x_3}$ 41
3.6 Compression of states represented by bits 42
3.7 Storing visited states using automata 43
3.8 Generating state-space with $\triangle$-mappings [Evangelista and Pradat-Peyre, 2005] 43
3.9 A part of reachability graph in Figure 3.4 using sequential model. The corresponding model is shown in 3.3 46
3.10 State change when tokens are created and/or deleted 47
3.11 Backtracking to expand State 4 in Figure 3.9 49
3.12 At depth d, the number of states is k^d. All states at depth δ are explicit . . . 55
3.13 A part of reachability graph in Figure 3.4 using tree algorithm 58

3.14 The difference form is determined using nearest explicit state, even if latter is a child (or descendent) of the former 58
3.15 Memory requirement decrease with increase in value of δ, the distance between two explicit markings. δ=0 means algorithm not used. Y-axis uses log scale. . 69
3.16 Delay increase with increase in value of δ. δ=0 when algorithm not used. . . 70
3.17 Delay increase and memory requirement decrease with increase in value of δ. Each curve is for a different value of δ as indicated by the legend. The six points on a curve correspond to six CPN models used. 70

4.1 Module for the protocol 79
4.2 Module for the Sender in Fig 4.1 80
4.3 Module for the Network in Fig 4.1 80
4.4 Module for the Receiver in Fig 4.1 81
4.5 Module for the Transmit in Fig 4.3 81
4.6 The declarations for modules in Figures 4.1-4.5 81
4.7 Module hierarchy for the example HCPN model. 82
4.8 The order of access for modules in the example HCPN model. 87
4.9 Reachability Graph for first row in Table 4.4. 89
4.10 Reachability Graph for second row in Table 4.4. 89
4.11 Module for the Network in Fig 4.1 98
4.12 The two possible cases when generating a parametrised reachability graph. . 105
4.13 The space occupied by access table increases with the number of enabling bindings. 108
4.14 The percentage of additional space occupied decreases with an increase in usage of parametrised reachability graphs. 109
4.15 The object model implementation using EMF. 112
4.16 HCPN model used for evaluation. 113
4.17 Time taken for model-checking the HCPN model shown in Figure 4.16. . . . 114

5.1 The identical components are identified and moved out. 122
5.2 A Coloured Petri-Net model with identical components. 124
5.3 The CPN model in Figure 5.2 with hierarchy installed using *substitution transition*. *Page1* is the supermodule while Subpage1(1) and Subpage1(2) are the two instances of submodule corresponding to the two substitution transitions. 124

5.4 The three algorithms constituting the two phases of the Lookup method. . . 127
5.5 The roadmap of the proposed solution. Blue indicates requirement that is addressed by the immediately following Algorithm (in red). 127
5.6 The example net for demonstrating the Lookup method. 129
5.7 The example net after Phase-1. 133
5.8 A possible set of groups in $\alpha_{example}$ for our example CPN. 135
5.9 α is not unique. Another possible set of groups in $\alpha_{example}$ for same CPN. . . 135
5.10 The hash-tables corresponding to a place. 137
5.11 The adjoining places of identical components to be added next 137
5.12 After adding the adjoining transitions to P2 and P4 140
5.13 After adding the adjoining places to component in Figure 5.12 140
5.14 The key-value pairs inserted into the hash-tables hashInIndex and hashOutIndex in first execution of Algorithm 14. 143
5.15 The key-value pairs inserted into the hash-tables hashInIndex and hashOutIndex in second execution of Algorithm 14. 143
5.16 Comparing two components using Algorithm 14. 144
5.17 The hash-tables in meta-group M_1 151
5.18 The hash-tables in meta-group M_2 151
5.19 The hash-tables in meta-group M_3 151
5.20 The hash-tables in meta-group M_4 151
5.21 The subnets in group V_5 151
5.22 The subnets identified in CPN model. 152
5.23 The two identical components (blue and brown) forming a group in example net 152
5.24 The equivalent hierarchical model obtained using the Clustering algorithm . . 153
5.25 Average time (in msec) taken by the Lookup algorithm for nets 1-7. The dotted-lines indicate the projection of curve on either axes. 157
5.26 Average time (in msec) taken by the Lookup algorithm for nets 8-12. The dotted-lines indicate the projection of curve on either axes. 157
5.27 Average time (in msec) taken by the Lookup algorithm for nets 13-17. The dotted-lines indicate the projection of curve on either axes. 158
5.28 Average time (in msec) taken by the Lookup algorithm for each net (1-17). The dotted-lines indicate the projection of curve on either axes. 158
5.29 Average time (in msec) vs Number of times Algorithm 14 is invoked for nets 1-7. 159

5.30 Average time (in msec) vs Number of times Algorithm 14 is invoked for nets 8-12. 159
5.31 Average time (in msec) vs Number of times Algorithm 14 is invoked for nets 13-17. 159
5.32 Average time (in msec) vs Number of times Algorithm 14 is invoked for all nets. Note that the curves overlap. 159
5.33 Processing time of Algorithm 14 and Algorithm 12 for nets 1-12. 161
5.34 Processing time of Algorithm 14 and Algorithm 12 for nets 13-17. 162

6.1 The architecture of proposed verification framework. 172
6.2 Spring configuration for a BPEL specification. 173
6.3 The class modified for underlying extension. 173
6.4 Bean for structured activities store reference to child activities. 174
6.5 The schema for a fragment of CPN model. 176
6.6 A CPN excerpt wherein the initial marking of a place is assigned from Java beans. 177
6.7 The instantiation and initialisation of classes for CPN excerpt in Figure 6.6. . 177
6.8 The Object Model for BPEL Activities 179
6.9 The template for top-level scope 184
6.10 The template for a basic activity 185
6.11 The template for an activity synchronised by links. 187
6.12 The template for items added to parent page. 189
6.13 The template for subpage corresponding to an underlying scope. 190
6.14 The template for flow activity. 191
6.15 The template for sequence activity. 192
6.16 The template for switch activity. 193
6.17 The template for inteface activities. 194
6.18 The template for while activity. 195
6.19 The template for assign activity. 196
6.20 The template for handler activities. 197
6.21 The modified template for basic-activity that can report faults. 199
6.22 Number of activities in BPEL specification. 201
6.23 The time taken for BPEL to CPN transformation. 201
6.24 Number of places and transition in the model rendered. 202

6.25 Number of states and edges in the state space of transformed models. 202
6.26 The time taken for generating first 50,000 states. 203

List of Tables

3.1 A comparison of solutions based on exhaustive storage 44
3.2 A comparison of the storage techniques for memory-reduction 44
3.3 Space occupied (in bytes) by first 500 states of CPN models (Λ) and percentage decrease in space (Δ). n and m are the number of places and tokens in these models. 68
3.4 Time (in msec) to generate first 500 states of CPN models (π) and percentage increase in time (η). n and m are the number of places and tokens in these models. 69

4.1 The compound places in HCPN model along with their initial marking 83
4.2 A comparison of the categories of existing solutions 85
4.3 A comparison of the existing solutions 85
4.4 The access-table for Sender 89
4.5 The values assigned to parameters by tokens 90
4.6 The tokens removed by *ReceivePacket* from its input places for each binding . 94
4.7 The tokens added by *ReceivePacket* to its output places for each binding . . . 95
4.8 The access-table of *Receive* from Tables 4.6 and 4.7 95
4.9 The access-table for Transmit 96
4.10 The access-table for Network 97
4.11 Tokens in input port-places that produce the enabling tokens in socket 98
4.12 The access-table for modified Network module 99
4.13 The entry in access-table for module *M2* 106
4.14 The access-table for *pg2* 115

5.1 A comparison of related reduction methods 126
5.2 Hash-table for places created in 2^{nd} pass 135

5.3 Hash-table for transitions created in 2^{nd} pass 135
5.4 Initial hashIn_{24} 139
5.5 Initial hashOut_{24} 139
5.6 hashIn_{24} after adding the adjoining transitions 139
5.7 hashOut_{24} after adding the adjoining transitions 139
5.8 Initialisation of hashInIndex is followed by a change in iList for each subnet in M_1 147
5.9 Initialisation of hashInIndex is followed by a change in iList for each subnet in M_2 148
5.10 Initialisation of hashInIndex is followed by a change in iList for each subnet in M_3 149
5.11 Initialisation of hashInIndex is followed by a change in iList for each subnet in M_4 150
5.12 The nets [Mukherjee, 2009a] used for analysing execution time of the Lookup method 160

6.1 A comparitive summary of related works. 171
6.2 The classes for compensation-handlers, fault-handlers, event-handlers and primary activity 180
6.3 The classes for BPEL structured activities 180
6.4 The use of three additional arcs in interface template 194
6.5 The random distribution functions offered by CPN tools 198
6.6 The difference between BPEL specifications used 200

Abstract

The last decade has seen a massive migration towards the service oriented paradigm. Companies have been actively exposing their software as services to be used by other applications over the web. Thereupon the ubiquity of the internet has allowed these services to be used from across the globe. Such state of affairs have resulted in 1) resolving the software interoperability issues, 2) increased re-usability of the code, 3) easy inter-application communications, and 4) significant cost reduction. These advantages have created tremendous business opportunities for companies offering web-services.

However, individual web-services seldom meet the business requirements of an application. Usually an application life-cycle involves interacting with several web-services based on its workflow. Considering that this might require 1) sharing data with multiple services, 2) tracking the response for each service request, 3) tracking and compensating the service failures, etc., usually a domain-specific language is used for service composition. Each service has an interface to outline its functionality and they are composed based on these interfaces.

Nevertheless, any error or omission in these exposed interfaces could result in a myriad of glitches in the composition and the overlying application. This is further exacerbated by dynamic service composition techniques wherein services could be added, removed or updated at runtime. Consequently service consuming applications heavily depend on the verification techniques to vouch for their reliability.

Traditionally software systems were verified by rigorously testing them against a set of test cases. The success of such techniques entirely depends on the methods used to decide on the test cases. Considering the complexity of contemporary systems, a comprehensive knowledge of the system is rare and test cases are often based on educated guesses.

The scope of applications based on service composition is rapidly expanding into critical domains where the stakes are high (e.g. stock markets, financial transactions). Consequently their reliability cannot be solely based on educated guesses. Considering that model-checking

techniques exhaustively verify a system, they need to be used in conjunction with testing techniques to further enhance the reliability. Model-checking [Clarke et al., 2000] is a formal method that has an unprecedented ability to endorse the correctness of a system. It involves modeling a system before verifying it for a set of properties using a model-checking tool. However it has hitherto been sparingly used because of the associated time and memory requirements. Consequently model-checking techniques are sparingly used in verifying a service composition. This is further exacerbated by the size of formal representations that are often too large for human comprehension.

This thesis proposes novel solutions to deal with these limitations in verifying a service composition. We propose a technique for modeling a service composition prior to verifying it using a model-checking tool. Compared to existing techniques that are ad-hoc and temporary, our solution streamlines the transformation by introducing a generic framework that transforms the composition into intermediate data transfer objects (DTOs) before the actual modeling. These DTOs help in automating the transformation by allowing access to the required information programmatically. The experimental results indicate that the framework takes less than a second (on average) in transforming BPEL specifications. The solution is made more appealing by further reducing the aforementioned time and memory requirements for model-checking. The additional reduction in memory is attributed to storing the states as the difference from an adjoining state. The reduction in time is realized by exploring the modules of a hierarchical model concurrently. These techniques offer up to 95% reduction in memory requirements and 86% reduction in time requirements. Furthermore, the time reduction technique is also extended to non-hierarchical models. This involves introducing hierarchy into a flat model in linear time before applying the time reduction techniques. As compared to other techniques, our method ensures that the transformed model is equivalent to the original model.

Chapter 1

Introduction

Service oriented architecture (SOA) based applications are built as an assembly of existing web-services wherein the component services can span across several organizational boundaries and have any underlying implementations. These services are invoked in some sequence based on the business logic and the workflow of the application. Such state of affairs have allowed alleviating software interoperability issues and catapulted SOA into the forefront of software-development architectures. The rapid inroads made by such applications can be attributed to their agility, maintainability and modularity.

1.1 The Problem

SOA-based applications require software components to be exposed as *services*. Each service has an *interface* that outlines the exposed functionality. Ideally an application is designed by discovering appropriate services using their interfaces and composing them. Such static compositions require the involved services to be perpetual and consistent throughout the lifetime of the application. Microsoft Biztalk [Vasters, 2001] and Oracle WebLogic [Jacobs, 2003] are among popular static composition engines.

However, existing web-services can break and newer (probably better) services can surface. Furthermore a change in business logic of an application during its lifetime might necessitate additional web-services to be composed dynamically. Such state of events have culminated in dynamic web-service composition. As compared to their static counterparts, an application based on dynamic composition is open for modification, extension and adaptation at runtime. Stanford's Sword [Ponnekanti and Fox, 2002] and HP's eFlow [Casati et al., 2000] are among the popular dynamic service composition platforms.

Nevertheless, dynamic composition [Zeng, 2003; Chan and Lyu, 2008] presents immense challenges. An enterprise application is expected to be void of any deadlocks, live-locks and conflicts. A static composition can be verified for these behavioural properties at design time. However, the verification for dynamically composed applications can only be done at runtime. This is further exacerbated by services that disobey their exposed interfaces at runtime. Considering the remarkable ingenuity of formal methods in runtime verification of traditional systems, they ought to be used for SOA based applications [Bayazit and Malik, 2005; Gan et al., 2007].

The reliability of SOA based applications require a lifelong verification of the corresponding service composition. This includes verifying a composition at design-time and monitoring its behaviour at runtime. The verification at design stage involves generating and scrutinising the entire state space of the composition for behavioural properties. Unless the underlying composition is altered at runtime, the application would always be in one of these scrutinised states. However, as shown in Figure 1.1, the application might reach uninvestigated states at runtime owing to the dynamic composition. These states could be reached if services in the target application are added, removed or updated. In order to verify the behavioural properties for runtime states, the model-checking should not terminate with design time verification. Instead it should continue at runtime to determine the uninvestigated states reached by the application and scrutinise them to verify the behavioural properties.

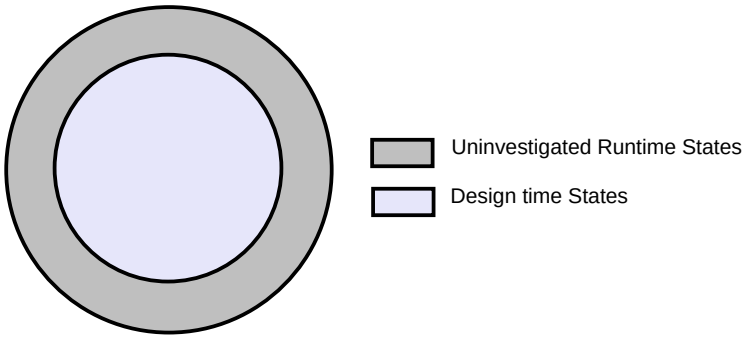


Figure 1.1: The additional states for runtime verification

Conventional techniques [Myers, 1979; Ammann and Offutt, 2008] cannot be applied for verifying a SOA based application because 1) the fault, if any, is mostly related to the business logic for service-composition rather than the source-code or implementation of underlying services; 2) even if an issue is found with the implementation of a service, the source code is usually not available for rectification; 3) even if the source code is available, it cannot be

immediately rectified as this might break probably thousands of other applications using this service. Furthermore, the reliability of conventional verification techniques were seriously undermined by the crash of Ariane 5 launcher [Clarke et al., 2000] and the deaths due to malfunctioning of Therac-25 radiation therapy machine [Rushby, 1989] in spite of rigorous software-testing. The teams investigating these disasters recommended using formal methods (FM) to complement testing as the former assures exhaustive verification of a system [Clarke et al., 2000; Rushby, 1989].

Formal methods have a remarkable ingenuity in warranting the safety of a system. They involve writing a formal description of the system under deliberation and analysing it to discover faults and inconsistencies. The formal description of a system is abstract, precise and complete [Hall, 2005]. While the abstractness allows a high-level understanding of the system, all inconsistencies and ambiguities in it are resolved in formulating a precise and complete description. Furthermore, the abstractness also allows ignoring the underlying architectural differences in the SOA based applications and analysing them like any other software application. Formal methods are often applied at the early stages of application development that involve requirement analysis, specification and high-level design.

The formal description for a SOA based application should comprise of the business logic for underlying service composition. Among all the domain-specific languages that were proposed for specifying the web-service composition, Business Process Execution Language for web-services (BPEL4WS or just BPEL) [Curbera et al., 2002; Andrews et al., 2003; Arkin et al., 2005] stands out as the de-facto industry standard. Unfortunately the overlapping constructs [Wohed et al., 2002] and the lack of sound formal or mathematical semantics [van der Aalst, 2003; Schmidt and Stahl, 2004] in BPEL do not allow it to be used as a formal description. These inconsistencies are the outcome of two conceptually contrasting languages (Web Services Flow Language (WSFL) [IBM, 2001] of IBM and XLANG [Thatte, 2001] of Microsoft) that were amalgamated to constitute BPEL [Schmidt and Stahl, 2004]. This necessitates transforming the textual specification of BPEL into a formal description prior to any formal analysis.

Unfortunately the existing solutions for formalising a BPEL specification are ad-hoc and temporary [Foster et al., 2003; Kang et al., 2007; Yang et al., 2005; Yi and Kochut, 2004]. Despite the myriad of modeling languages available (e.g. Promela, Petri Nets, Automata, Process Algebras), these solutions specifically target a particular language. In pursuit of a generic solution, we transform a BPEL specification into an intermediate specification before the actual formalisation. In software engineering, Data Transfer Objects (DTOs) are

commonly used design patterns for storing and transferring data [Crawford and Kaplan, 2003]. Therefore we use DTOs to store the generic intermediate specification, wherein each BPEL activity is mapped to a separate DTO. These DTOs can thereupon be transformed into any modeling language.

However, model-checking techniques usually have an associated time and memory requirement that far outweighs the available resources. Therefore software developers often skip formal-methods to meet software budget and deadlines. Consequently it is necessary to address these issues prior to using it for verifying SOA based applications. As illustrated in Figure 1.2, the proposed technique for verifying a BPEL specification is preceded by memory and time reduction procedures for model-checking [Clarke et al., 2000], a widely used formal method. The time and memory costs for model-checking are not independent as any effort in reducing the memory is accompanied by an increase in delay [Evangelista and Pradat-Peyre, 2005].

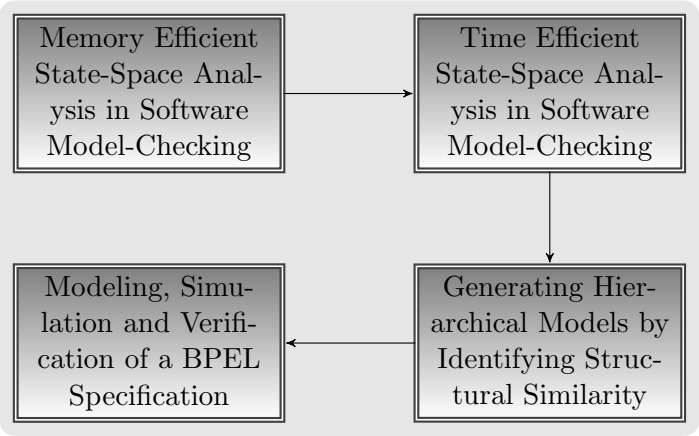


Figure 1.2: The chronology of research.

Although it is possible to model-check a system irrespective of the size and orientation (i.e. flat or hierarchical) of its model, it is important to recognize the advantages of a hierarchical and succinct system representation. A hierarchical model is easy to draw and practical to analyse and maintain [Jensen and Kristensen, 2009]. Analysing a system model, in turn, might assist in accomplishing additional objectives like identifying the overall architecture of the system, understanding its dependencies, visualising the flow of information through it, identifying its capabilities and limitations and calculating its complexity [Christopher, 2003]. We recognize the importance of a succinct and modular system representation and propose techniques to introduce hierarchy into a flat model. As illustrated in Figure 1.2, this

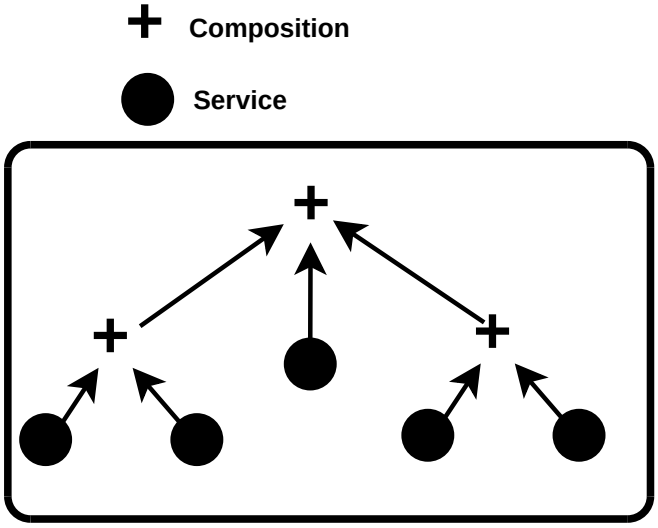


Figure 1.3: A SOA based application constitutes a hierarchy of services.

technique immediately precedes the procedure for model-checking a SOA based application.

1.2 Statement of the Problem

SOA based applications are prone to failures and inconsistencies owing to multiple *single point of failures* (SPOFs) [Chakraborty et al., 2002; Hu et al., 2005]. A SPOF is an element or part of a system whose failure leads to a catastrophic system crash. Such a crash could either terminate the service or instigate it to exhibit erroneous behaviour. Typical SPOFs in a service composition could be 1) the composition itself, and 2) any component service that is also created by composing services. This is essentially because the reliability of the constituent services do not guarantee the reliability of the composition. Consequently irrespective of the efforts in validating the services, their composition remains a SPOF. The problem is further exacerbated by dynamic service compositions wherein services in a composition could be added, removed or replaced at runtime. Considering that a SOA based application constitutes a hierarchy of services, a failure at any level could break the application. This is illustrated in Figure 1.3

Although model-checking techniques can be used to exhaustively verify a service composition and determine the SPOFs, they are sparingly used due to the associated time and memory requirements. Model-checking involves scrutinising the reachable states of a system for a set of predefined undesirable properties. However, modern software systems have very

large state-space owing to their complex and concurrent components. Consequently the time required in scrutinising each of these states is massive. This is further aggravated by systems that repeatedly reach one or more states during their course of execution. In the absence of a mechanism to detect duplicate states, model-checking could last forever. This is prevented by storing the states generated hitherto in memory and comparing them to any state reached subsequently. The massive memory requirement is attributed to storing the large state-space of the system.

The formal representation (or model) of a service composition can also be used to determine the SPOFs. This is essentially because a formal model is unambiguous owing to its mathematical semantics. However, the representation could be massive for a sufficiently complex composition. Consequently it might be impossible for a human modeler to analyse and determine the SPOFs.

The last three decades have seen extensive research on model-checking techniques [Grumberg and Veith, 2008]. Most of the research has been driven by the aforementioned state-space explosion problem [Christensen et al., 2001] associated with it wherein an overwhelmingly large number of states need to be checked to verify the system. Although numerous techniques have been proposed to deal with this problem [Evangelista and Pradat-Peyre, 2005; Holzmann, 1997; Christensen et al., 2001; Visser, 1996], it is still open for further improvement.

1.3 Research Questions

The tremendous growth of web-services over the last decade has necessitated addressing the problems with formal methods and using them to enhance the reliability of services composition. The consequence of the problem is two-fold: 1) the memory costs for storing the states increase with the size of state-space, and 2) the delay in generating the state-space increases with the size of state-space. These problems are the basis of the first two major research questions that are addressed in this thesis. The remaining questions deal with the installation of hierarchy into a model and verification of a BPEL specification.

How can we reduce the memory costs otherwise involved in model-checking a service composition?

In order to verify a composition, it needs to be formalised and subjected to a model-checking tool. The tool generates the state-space of the system and scrutinises them for undesirable

properties. During state-space exploration, it is possible for a state to be generated more than once. To prevent analysing the same states repeatedly for undesirable properties, it is necessary to remember the states already explored by storing them in memory. This also ensures termination, a condition where no new states could be generated. However, model-checking is plagued with state-space explosion problem, often resulting in gigantic number of states. This causes manyfold increase in memory costs, as each new state has to be stored. This question looks into ways of reducing this memory requirement.

How can we reduce the time delay otherwise involved in model-checking a service composition?

State-space analysis of a service composition is done by generating its reachability graph wherein each node in latter corresponding to a state of the former. However, the ever-increasing intricacies in contemporary SOA based systems snowballs the reachability graph to contain a gigantic number of states. This phenomenon, better known as the state-space explosion problem [Christensen et al., 2001], leads to exorbitant delays in producing and analysing the state-space. This is further undermined by the necessity of storing states that were generated hitherto and comparing them to any state produced henceforth. Regardless of the numerous ingenious algorithms proposed for an efficient storage and comparison of states, there is always an associated time overhead. This question looks into ways of reducing this delay.

How can we install hierarchy into a flat model to make it exponentially more succinct?

A service composition needs to be modeled using one of the several available modeling languages prior to generating its reachability graph. This being a tedious and error-prone activity, several techniques have been proposed to auto-generate a formal representation of the software system under consideration [Chen and Cui, 2004; Fu et al., 2004]. However, the primary objective in auto-generating a model is to produce the input for a model-checking tool. Consequently there is limited incentive in enhancing the human understandability of the rendered model by introducing modularity and hierarchy. The formal model for a service composition might assist a human modeler in accomplishing additional objectives like identifying the overall architecture of the composition, understanding its dependencies, visualising the flow of information through it, identifying its capabilities and limitations and calculating

its complexity [Christopher, 2003]. However, the flat model produced by auto-generating techniques pose a serious challenge in accomplishing these objectives. This question looks into ways of introducing hierarchy into a flat model in order to make it exponentially more succinct.

How can we model, simulate and verify a BPEL specification?

Being loosely coupled systems, the safety and reliability of SOA based applications entirely depend on the precision of service descriptions. Consequently any implicit assumption or unforeseen usage scenarios can lead to undesirable forms of interactions, such as a deadlock or race condition [Sloan and Khoshgoftaar, 2009]. This is further exacerbated by dynamic service composition wherein services could be added, removed or updated at runtime. Considering that business process execution language (BPEL) is the de-facto industry standard for service composition, the inconsistencies and ambiguities in it further aggravates the problem. This question looks into ways of modeling, simulating and verifying a BPEL specification.

1.4 Limitations of Existing Solutions

In this section we underline the limitations of existing solutions. These limitations are further discussed in chapters concerning individual research questions.

Memory efficient state-space analysis in software model-checking

In spite of extensive research over the last three decades, memory efficient state-space analysis techniques are still open for further improvements. Most of these existing solutions can be classified into either of 1) Exhaustive storage [Schmidt, 2003; Evangelista and Pradat-Peyre, 2005; Holzmann, 1997]; 2) Partial storage [Christensen et al., 2001]; and 3) Lossy storage [Visser, 1996; Holzmann and Puri, 1999] techniques. While the exhaustive storage techniques compress and store each state in the state-space, partial-storage techniques store only a subset of these states. Lossy storage techniques differ from exhaustive techniques in using compression algorithms that are not reversible. The exhaustive techniques are characterised by the compression algorithm used to encode a state (e.g. state collapsing [Visser, 1996], recursive indexing [Holzmann, 1997], very tight hashing [Geldenhuys and Valmari, 2003], sharing trees [Grégoire, 1996], difference compression [Parreaux, 1998]). As indicated previously, each of these techniques have an associated time delay. The solution proposed in [Evangelista and Pradat-Peyre, 2005] triples the delay when reducing the memory costs by

95% and is therefore it is considered better than other existing techniques. The partial technique uses specific algorithms (e.g. state-space caching) to determine the states that could be safely deleted from memory. Exhaustive techniques are by far the most generic technique and are widely used. However, as mentioned earlier, all the techniques are open for further improvements. The improvements include 1) further reducing the memory requirements for model-checking, and 2) reducing the associated time overhead.

Time efficient state-space analysis in software model-checking

The time efficient state-space analysis techniques have been less extensively researched as compared to memory efficient techniques. This possibly indicates a greater tolerance to delays in model-checking. All existing solutions for reducing the time requirement for model-checking can be categorised as either of 1) Partial order reduction [Evangelista and Pradat-Peyre, 2006; Kristensen and Valmari, 1998]; 2) Symmetry based reduction [Elgaard, 2002]; or 3) Modular state-space generation [Christensen and Petrucci, 1995]. Partial order techniques involves determining the *stubborn-sets* (i.e. a set of transitions such that a transition outside the set cannot effect their behaviour) and executing only the enabled transitions in each set. However, the problem of deciding if a set of transitions is stubborn at a state is at least as hard as the reachability problem [Clarke et al., 2000]. The symmetry method exploits the presence of any symmetrical components in a system that exhibit identical behaviour and have identical state graphs. The sub-graphs of these components in the reachability graph of the entire system are usually interchangeable with some permutation of states. However, it is difficult to determine a sub-graph whose permutations would produce other sub-graphs (known as the *orbit problem* [Clarke et al., 1998; Emerson and Sistla, 1996]). Furthermore these techniques are void for models that lack symmetry. Modular state-space generation involves generating the reachability graph of each module independently and then composing them to generate the reachability graph for an entire model. Although modular techniques look more promising, most of the existing solutions are dated [Christensen and Petrucci, 1995].

The reduction in size of a formal model

Most of the existing solutions reduce the size of a model by transforming it based on a set of proposed postulates. The transformations proposed in [Berthelot, 1986] aim to reduce the size of Petri-Net models by merging two or more of its places or transitions based on certain

conditions. The implicit place simplification and pre and post agglomeration of transitions are the most frequently used transformations. These transformations were extended for coloured Petri-Nets in [Haddad, 1990] and [Evangelista et al., 2005]. Although these transformations preserve several classical properties of nets (like boundedness, safety, liveness etc) and also reduce the number of reachable states when performing state-space analysis, the transformed model might not be equivalent to the original model. Consequently the analysis of the reduced model would be incomplete.

The verification of a BPEL specification

The existing solutions for formalizing a BPEL specification involves a transformation into either of 1) Petri Nets / Coloured Petri Nets [Kang et al., 2007; Yang et al., 2005; Sloan and Khoshgoftaar, 2009; Stahl, 2005], 2) Process Algebras [Ferrara, 2004], 3) Abstract State Machine [Fahland, 2005; Fahland et al., 2005] or 4) Automata [Arias-Fisteus et al., 2004; Fu et al., 2004]. The problem is addressed by generating a model for each BPEL activity using one of the aforementioned modeling languages. Thereafter the users are required to scan the BPEL specification and replace each activity with its corresponding formal-model. Apart from being a cumbersome process, such an exercise is error-prone and time-consuming. Although there exist some solutions that automate this translation, they do not consider BPEL's most interesting and complicated activities like *eventHandler* and *links* [Fu et al., 2004]. In spite of being feature complete, the models obtained using [Stahl, 2005] are bulky and error-prone owing to the plain-vanilla Petri Nets used. The abstract state-machine based solutions are also feature complete. However, they lack adequate tool support for simulation and verification.

1.5 Contributions

This section highlights the research contribution made in this thesis to address the various research questions described in Section 1.3.

A memory efficient state-space analysis technique

A technique is proposed to reduce the memory costs otherwise involved in model-checking a service composition by storing states as the difference from one of the neighbouring states. Asserting that "the change in a state is always smaller than the state itself", this technique brings in up to 95% reduction in memory costs. Storing the states in difference form results

in the reduction of memory requirements. Based on the neighbouring state used to calculate the difference, the technique is further divided into two related models, 1) *Sequential model* stores a state as how different it is from its immediately preceding state, and 2) *Tree model* stores a state as how different it is from its nearest state in explicit form. The solution is based on exhaustive storage technique discussed in Section 1.4. Aforesaid reduction allows model-checking in a machine with only 5% of the memory needed otherwise. Consequently the advantage is twofold: 1) only 5% of the physical memory is required to validate the composition, and 2) as more states can now be stored in a memory of same size, the chances of complete state-space analysis of a composition is high. The proposed compression algorithm provides a 95% reduction in memory requirements with only twice the delay. Other solutions [Evangelista and Pradat-Peyre, 2005; Holzmann, 1997] incur considerably larger delays and offer significantly less memory reduction.

A time efficient state-space analysis technique

This method reduces the time requirement for model-checking a service composition. It necessitates the composition to be formalised as a hierarchical model. A hierarchical-model consists of a set of inter-dependent modules and the offered reduction in delay is attributed to the concurrent exploration of all such modules in a hierarchical model and exposing the outcome using special data-structures. These data structures, known as *Parametrised Reachability Graph* (PRG) and *Access-tables*, act as a repository of corresponding module behaviour and a module can use these data-structures to determine the behaviour of any other module without actually executing it. In addition to concurrency, exposing such module behaviour repositories prevent executing a module more than once and thereby help in reducing the delay. The dependency of other modules on a module is injected into its repository using parameters. Later these parameters are assigned specific values to obtain the corresponding reachability graph for the hierarchical-model. The proposed technique offers a time reduction of 86% in generating the first 25,000 markings. Other solutions [Evangelista and Pradat-Peyre, 2006; Kristensen and Valmari, 1998] offer significantly less reduction in delay. Furthermore the proposed solution has less stringent prerequisites (i.e. a hierarchical model) as compared to other techniques that necessitate stubborn sets or symmetry in the model.

A technique for reducing the size of a model exponentially

This technique allows an exponentially more succinct representation of a service composition by embracing the notion of hierarchy. A hierarchical model consists of a set of modules wherein each module represents a system component. In such a setup, the module for a high-level component refers to its underlying components using their module name or reference. This avoids the "blow-up" in including the actual representation of underlying components. Furthermore, the benefits increase with each additional high-level component sharing an underlying component. Consequently the obtained model would be exponentially more succinct owing to the notion of hierarchy introduced. The proposed solution establishes hierarchy after identifying the set of structurally similar components in a model. The experimental results indicate that this takes linear time. Furthermore, the time also depends on the number of identical components in the model. The solution is generic and can be applied for any modeling language that defines the semantics of hierarchy and structural similarity.

A technique for modeling, simulating and verifying a BPEL specification

We propose a verification framework to formalise a BPEL specification by transforming it into an XML based formal model. The existing solutions utilise ad-hoc techniques to formalise a BPEL specification into a specific modeling language. However, the massive growth of SOA based applications in recent years have necessitated the streamlining of BPEL formalisation. In pursuit of generalising the transformation, the specification is initially transformed into intermediate DTOs. This is done by extending the *Spring framework* to represent each BPEL activity using a *Java bean*. Spring helps in significantly automating the creation of intermediate DTOs. The framework instantiates the beans corresponding to activities in a BPEL specification and injects the dependencies to yield a *bean-factory*. This bean-factory contains all the information required to construct a formal model.

As mentioned previously, the generic intermediate specification could be transformed into any modeling language. However, to demonstrate the actual formalization, these DTOs are transformed into an XML based formal model (e.g. Coloured Petri nets (CPN) [Jensen and Kristensen, 2009]). This is done using *Java Architecture for XML Binding (JAXB) 2* APIs that offers a practical, efficient and standard way of mapping between XML and Java code. However, the JAXB 2 APIs require an XML mapping for each BPEL activity. It uses the corresponding schemas to transform the bean-factory into a formal model. Consequently a

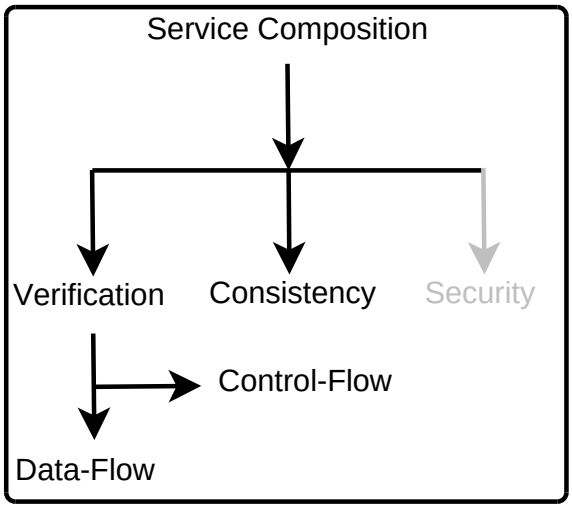


Figure 1.4: The scope of research.

Coloured Petri-net based template[1] is proposed for each BPEL activity. Considering that CPN models 1) offer hierarchical semantics, 2) are visually expressive, 3) can be both simulated and verified, 4) are XML documents like BPEL specification, 5) have extensive tool support, and 6) support inbuilt and user defined data-type, they are selected as the target for proposed transformation. Furthermore, an object-model has been proposed to determine the relationship among BPEL activities. This object model is used to create the Java-beans corresponding to BPEL activities. In addition, the proposed CPN templates exploit any hierarchy in object model to reuse a parent template for its child activities after any required customization. The solution is feature complete and extendible.

1.6 Scope of Research

Figure 1.4 defines the scope of our research. We investigate the issues with verification and service consistency and propose novel solution. However we neither identify the security issues with composition nor address them.

Verification involves determining the correctness of service composition and its conformance to the desired requirements and functionalities. This requires scrutinising the data-flow and control-flow of a composition for errors and inconsistencies. The data flow exhibits how data is exchanged within a service and among the involved services. Verifying the data-flow ensures correct assignments among services at runtime. This is excessively important

[1] templates are part of a complete formal model

for dynamic service compositions wherein the constituent services change at runtime. The control-flow shows the workflow for service invocations. Verifying the control-flow ensures the correctness of obtained application.

Service consistency involves determining the compliance between a set of Web Service interfaces. Since such interfaces are specified using Web Service Description Language (WSDL), the latter is used to check for conformance.

Security for a service composition deals with protecting the client information, certifying the identity of services, providing authentication mechanisms for the network infrastructure, determining the compliance of services with a security policy etc.

1.7 Thesis Structure

The remainder of the thesis is organised as follows:

Chapter 2 provides the necessary background knowledge to ensure a better understanding of the concepts applied herein. This includes a run-through of model-checking, Coloured Petri nets, Hashing, BPEL activities, Spring framework and Java Architecture for XML binding (JAXB) 2 APIs.

Chapter 3 presents two techniques for reducing the memory costs otherwise involved in model-checking by storing states as the difference from one of the neighbouring states. Theoretical evaluation and experimental results indicate a significant reduction in memory requirements.

Chapter 4 proposes a novel method to reduce the time requirement for model-checking a hierarchical model by exploring its inter-dependent modules in parallel. These dependencies are stored as parameters in special data-structures. On assigning specific values to these parameters, these dependencies are resolved and the envisioned reachability graph is obtained. The experimental results indicate a significant reduction in time requirement.

Chapter 5 proposes a technique to install hierarchy into a flat model by identifying the structural similarity. The technique is based on decrease and conquer technique wherein the bigger problem is broken into smaller problems and the solution to smaller problems are combined to solve the original problem. As compared to existing techniques, the rendered model is equivalent to the original model.

Chapter 6 proposes a verification framework to formalise a BPEL specification by transforming it into an XML based formal model. This is done by extending the Spring framework and using the JAXB 2 APIs. In addition, we determine a hierarchical relationship among

BPEL activities to enhance the efficiency of this transformation. This framework 1) is extendible, 2) has significantly small transformation time, and 3) can be used for existing techniques.

Finally, the thesis concludes in Chapter 7 after summarising the main contributions in this thesis and listing the possible directions of future research.

Chapter 2

Background

In this chapter we present the methods that form the foundations of the proposed solutions. Section 2.1 covers the fundamentals of model-checking techniques that are used in Chapters 3, 4 and 6. Model-checking techniques are used in this thesis to propose a framework for verification of service compositions. Section 2.3 introduces Coloured Petri net formalism that is used in Chapters 3 and 4 for proposing the time and memory reduction techniques. Thereafter the basics of Business Process Execution Language (BPEL) are summarized in Section 2.4 along with its activities. BPEL is the de-facto industry standard for service composition and is verified in Chapter 6 to enhance its reliability. Section 2.5 introduces the basics of Spring framework that is extended in Chapter 6 to automatically transform a BPEL specification into a hierarchy of intermediate DTOs. Finally Section 2.6 covers Java Architecture for XML binding APIs that transform the intermediate DTOs into an XML based formal model.

2.1 Model-checking

Steven C. McConnell, one of the most influential figures in software development industry, observed [McConnell, 2004] 1) an industry average of 15-50 errors per 1000 lines of delivered code, 2) about 10-20 defects per 1000 lines of code in Microsoft applications, and 3) Formal development methods, peer reviews, and statistical testing helped reducing this to 0 per 500,000 lines of code.

Traditionally a rigorous testing phase establishes the reliability of software systems [Beizer, 1990]. This involves executing the system with a set of inputs to verify if it behaves expectedly. The success of such techniques entirely depend on the methods used to decide on

test cases. Usually a thorough knowledge about the system is required to decide on such cases. However, considering the complexity of modern systems, this is often not the case. Consequently most systems are tested based on *educated guesses* on the basis of any partial understanding of the system.

Model-checking (MC) [Clarke et al., 2000] is a formal method for automatic verification of software systems that offers distinct advantage over conventional testing and simulation techniques. It is a rigorous technique wherein all possible behaviours of the system are scrutinised exhaustively. Consequently if there is a problem, model-checking will detect it. It is being increasingly adopted as a standard procedure for quality assurance of software systems [Merz, 2001]. Contrary to traditional software systems that compute some result from the given input values, a modern software system maintains an ongoing interaction with its environment and is therefore known as *reactive system* [Harel and Pnueli, 1985; Manna and Pnueli, 1992]. Because of the non-deterministic nature of such systems, any amount of testing is grossly inadequate to estimate their reliability [Schneider, 2004]. Consequently MC is progressively complementing standard testing procedures as a part of software development. It involves three basic steps: 1) Modeling the system, 2) Specifying the properties to be verified and 3) Verifying the properties in all possible states of the system. Initially a model checker requires formal design of the system and the properties to be verified. It then explores the state-space of the system to find a state[1] which violates the given properties, where state-space is the set of states reachable from initial state. If a violating state is found, it is returned as a counterexample. Otherwise, the model checker returns 'yes', implying that the properties are satisfied by all reachable states of the system.

Figure 2.1 illustrates the process of model-checking. As a prerequisite, it requires a formal model and the specification to be verified. Considering the time and memory constrains, the former is often an abstraction of the system to be verified. The specification states the properties that the target system must satisfy. This is often specified using temporal logic. The model checker verifies if the properties stated in the specification is satisfied for all possible execution paths of the system. In case the specification is not satisfied by the system, the model checker returns an error trace.

Unfortunately the number of reachable states of any non-trivial system is immensely large. Since a model checker is required to explore and scrutinise each of these states for the set of desirable properties, model-checking is associated with a massive delay. Furthermore the unique states explored hitherto by the model checker needs to be stored in memory to

[1] 'Marking' is sometimes used synonymously to a state

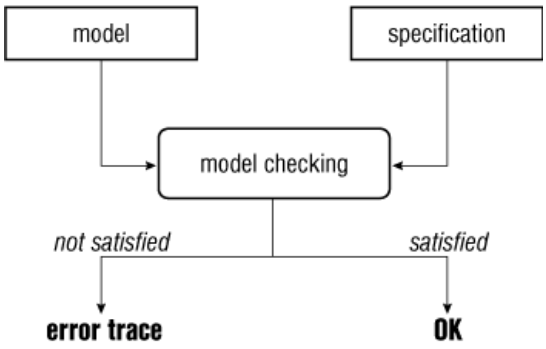


Figure 2.1: The process of Model-checking

prevent repeatedly exploring the same state. Irrespective of the space occupied by each state, model-checking also has a high memory requirement.

Model-checking has been used for verifying service compositions in this thesis because:

- Model-checking offers exhaustive verification of the system under deliberation. Consequently the offered enhancements in reliability and safety outweigh any alternate techniques.
- As discussed in Chapter 1, model-checking allows a lifelong verification of service compositions. This is done by verifying a composition at design-time and monitoring its behaviour at runtime.
- Model-checking requires creating an abstract representation of the system before verifying it. The experience in generating this representation leads to a better understanding of the system.
- Model-checking has excellent tool support [Holzmann, 2003; Ratzer et al., 2003] that ensures an automatic verification.
- It does not require proving theorems.

2.2 Petri Nets

As observed in the previous section, model-checking necessitates a formal representation of the system prior to verifying it. This essentially involves creating a formal model of the system using any of the available modeling languages (e.g. Promela, Petri-nets, Process Algebras, Automata). However, the modeling language used significantly influences the properties of

the rendered model. For instance the concurrent constructs of a system cannot be mapped into its model using automata.

Petri net (PN) is a mathematical modeling language and a graphical tool for description and analysis of concurrent systems. It is a directed bipartite graph that consists of places (circles) and transitions (rectangles) connected by arcs. The tokens (black dots) in the places define the state of a Petri net. Events associated with a transition move the tokens between its adjoining places along the arcs. It has been widely used as a design language for specification of intricate workflows [van der Aalst, 1998] owing to the graphical nature, expressiveness, formal semantics and analysis techniques.

Figure 2.2 illustrate a Petri net model and its execution. The circles are known as places while the rectangles are known as transitions. A transition T is enabled if each of its input places P contain Z tokens, where Z the weight on arc connecting P to T. Consequently the transition T in Figure 2.2 is enabled. When it fires, it removes Z_{in} tokens from each input place P_{in} and adds Z_{out} tokens to each output place P_{out} where 1) Z_{in} is the weight on arc connecting P_{in} and T, and 2) Z_{out} is the weight on arc connecting T and P_{out}.

Petri nets offer several advantages over other modeling languages:

- They have a graphical representation that is intuitive and easy to understand.
- They can model both states and events. While places and tokens are used to model states, transitions are used to model events.
- The can be both simulated and verified.
- Model-checking using Petri nets and its extensions have been extensively researched in the last three decades. This has led to a wide array of tools and techniques [Evangelista, 2005; Schmidt, 2000; Holzmann, 2003; Ratzer et al., 2003].
- Petri nets have strong mathematical basis [Murata, 1989].

However, the complex and concurrent components in modern software systems have led to an exorbitant increase in size of Petri net representations. Consequently a range of extensions have been proposed that allow a convenient representation of Petri nets [Jensen and Kristensen, 2009; Fehling, 1993; Lakos, 1995]. These are known as *high-level* Petri nets and are widely used.

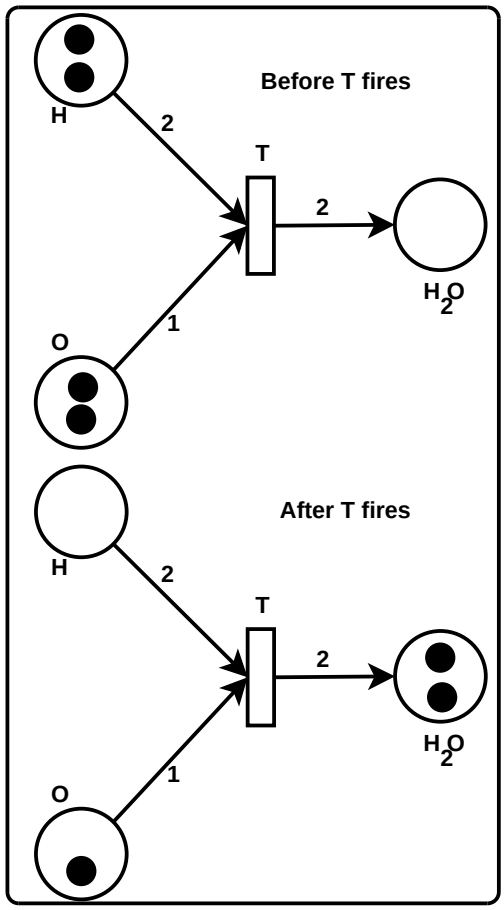


Figure 2.2: A Petri-net model [Murata, 1989].

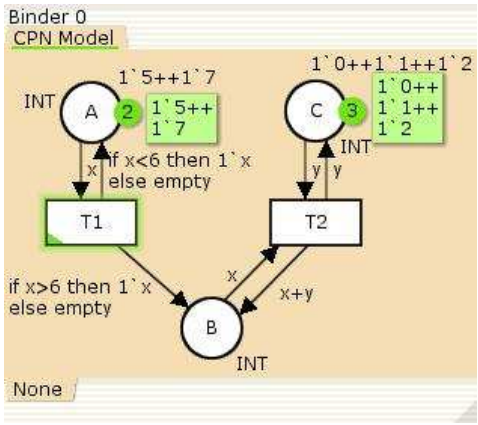


Figure 2.3: A coloured petri-net model. Variables X and Y are of type INT.

2.3 Coloured Petri Nets

Formal representations using plain vanilla Petri net are often significantly large. Consequently they are difficult to draw and impractical to analyse and maintain. As observed previously, a range of extensions have been proposed that allow a convenient representation of Petri nets [Jensen and Kristensen, 2009; Fehling, 1993; Lakos, 1995]. These extensions offer all the advantages of Petri nets discussed earlier. Furthermore, a formal representation using any of these extensions can be transformed into an equivalent Petri net.

A Coloured Petri-Net (CPN) [Jensen, 1996] is a Petri-Net extended with programming constructs. The concurrent constructs of the latter are supplemented with data-definition and data manipulation constructs of programming languages. CPN is used for design, specification, simulation and verification of systems. In contrast to PN, each token in CPN has an attached data value. The datatype of this value determine the colour of token. All tokens in a place must be of the colour specified by the colour set of that place.

Figure 2.3 shows a CPN model with 3 places (the circles) and 2 transitions (the rectangles). Place A has 2 tokens (indicated by 2 in the circle next to it) `1‘5` and `1‘7`, where `1‘5` implies that there is 1 token with integer value 5. The tokens in a place are listed next to it and they are separated by ‘`++`’ symbol. The text “`1‘0++1‘1++1‘2`” near place C implies that it has 1 token with value 0, 1 token with value 1 and 1 with value 2. Each place also has its colour (or datatype) inscribed next to it. In this model, all places have colour INT and hence they can only have integer tokens.

A place and a transition are connected by an arc. Transition `T1` in Figure 2.3 has an input

arc from place A and two output arcs to places A and B. When `T1` executes, it removes tokens from input place A and adds tokens to output places A and B. The tokens removed/added is found by evaluating the arc inscription. As all these arc inscriptions are defined in terms of variable `x`, they can be evaluated by assigning an appropriate value to `x` and this is called *binding of* `x`. The binding of `x` is decided after considering the tokens in input place and the inscription of input arc. In case of `T1`, input place A has tokens 1‘5 and 1‘7 while the input arc inscription is `x`. Hence the only possible values of `x` for which `T1` is enabled are 5 and 7. When `T1` fires with `x` bound to 5, token 1‘5 is removed from A and added back to A. No token is added to place B as the `if` condition is not satisfied. When `T1` fires with `x` bound to 7, token 1‘7 is removed from A and added to place B. The bindings for which `T2` is enabled can be determined identically.

The extensive use of Coloured Petri nets in this thesis could be attributed to the following benefits [Jensen, 1996]:

- All the aforementioned advantages for Petri nets are applicable for CPNs.
- They natively define hierarchical semantics for better representation.
- They have extensive tool support for drawing, editing, simulation and formal verification [Jensen et al., 2007].
- CPN models are XML based and therefore can easily be imported, exported and edited using third party applications.
- The structure of a CPN model is defined using a DTD [Westergaard et al., 2005]. Consequently the models could be easily checked for the validity of their structure.
- They offer various programmable elements that significantly reduce the size of formal models.

2.4 Business Process Execution Language (BPEL)

In recent years, companies have been actively exposing their software as a service to be used by other applications over the network. These services have unearthed tremendous business opportunities by exploiting the ubiquitous nature of web. They adhere to the associated service description irrespective of the underlying implementation.

However, as shown in Figure 2.4, these services by themselves might not be very useful. In the illustrated scenario, a customer needs to make several exclusive web-service calls to

complete a shopping. Furthermore, he needs to make them in some specific order wherein the reference from previous call is used in succeeding invocation. Apart from being a cumbersome and error prone process, the illustrated scenario also exposes the underlying business logic of the application.

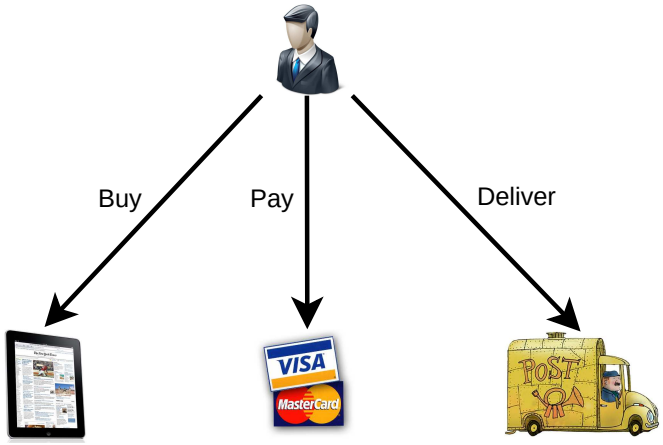


Figure 2.4: Using individual exposed services for online shopping.

Figure 2.5 illustrates an alternate scenario wherein the business logic is encapsulated within the application. The customer only needs to provide the payment and address details to finish the shopping. However, this necessitates composing the involved web-services based on the business logic of the application. Such state of affairs have resulted in the genesis of a large number of service composition languages. In recent years, BPEL has emerged as the de-facto language for composing web-services.

A BPEL process specifies the exact order in which the constituent web-services should be invoked. When calling a service, the information sent (or the result returned) could be read from (or stored into) a variable. Consequently the result from services called hitherto could be used for any forthcoming calls. It also supports conditional and iterative invocation of services. Furthermore it offers handlers that execute when an event occurs.

Although it is possible to compose services using a programming language (e.g. Java or C♯), there are specific advantages in using BPEL:

- Contrary to popular programming languages (like C♯), BPEL is platform independent. Therefore it can be used to compose services regardless of the platform used in creating them (i.e. Java, PHP, C++, C, Ruby, etc.).

- Business processes often require asynchronous interactions with high turnaround time.

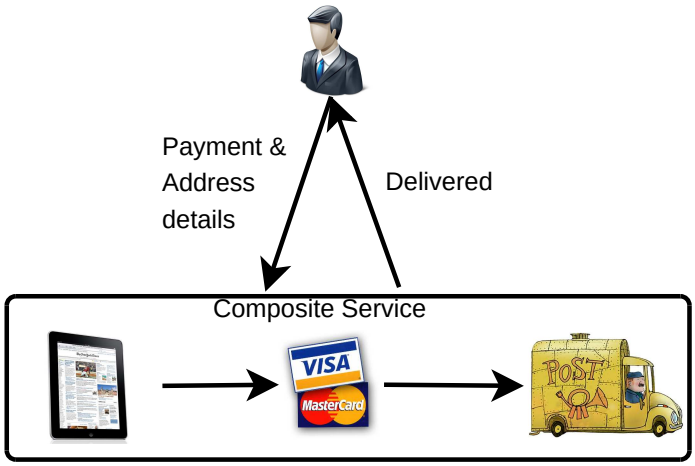


Figure 2.5: Using service composition for online shopping.

The BPEL activities have been created to handle such scenarios better than any programming language.

- It is easier to introduce concurrency into a BPEL process than most other programming languages.
- They are XML based and consequently can easily be imported, exported and edited using third party tools.

A BPEL specification consists of steps known as activities. These activities are further classified as *Basic activities*, *Structured activities* or other activities. Some important BPEL activities are as follows:

invoke and receive

The BPEL activity `invoke` is used to trigger a Web-Service. It has five attributes, `partnerLink`, `operation`, `outputVariable`, `inputVariable` and `portType`. The attribute `partnerLink` gives information about the role of a partner web-service. In case of a synchronous invocation, the role of partner web-service is to provide an `operation` which takes `inputVariable` as input and returns the result in `outputVariable`. In case of an asynchronous invocation, the `operation` takes `inputVariable` as input, but the result is returned using *callback*. We need a `receive` activity to fetch the result. Hence for asynchronous `invoke`, the role of partner is to provide `operation` while the role of BPEL is to receive result.

BPEL activity `receive` waits for an incoming message, either from a client to start BPEL process, or from a partner in case of *callback*. The attributes of `receive` are same as for `invoke`, except that it has a single variable to receive message. It also has an extra attribute `createInstance`. When this is set to "yes", a new instance of BPEL process is created. In case of a synchronous `receive`, the result is retuned to client using `reply` activity. All attributes of `reply` activity must be same as corresponding `receive`, except that the variable now has result to be returned.

These activities allow a BPEL process to interact with its constituent web-services. The advantages that these activities provide over conventional programming languages are as follows:

- An operation specified in a WSDL can be easily invoked using the `operation` attribute of invoke activity.
- The schema for input and output messages of an operation can be used to define the aforementioned variables.
- Both synchronous and asynchronous interactions are natively supported.

flow and sequence

BPEL `flow` activity defines a set of subtasks or sub-activities that must execute concurrently. It is also used to define guarded links. It terminates when the final subactivity in it completes. BPEL activity `sequence` defines a set of sub-tasks or sub-activities which are to be executed in specific order. This order is usually same as the order of occurrence of sub-activities within sequence. As with flow, a sequence activity terminates when the final subactivity in it completes.

Flow and Sequence activities offer an easy mechanism to control the flow of business processes.

link

BPEL activity `link` is used to synchronize concurrent subtasks of `flow` activity. Each link is identified by a unique `name` and has a `source` activity. It also has one or more `target` activities. Each link can have at most one subtask of flow as its `source` and one or more other subtasks as `target`. The `target` activities for a link can execute only when its `source` activity has finished execution. The `source` for a link might have `transitionCondition`

attribute specified, which evaluates a boolean expression. The result of this expression is sent to `target` activities for this link. If no `transitionCondition` is specified, *true* is sent by default.

while

BPEL activity `while` is used to repeat the enclosed activities until the boolean condition specified is no longer true. The condition is specified using `condition` attribute. The condition is evaluated before each iteration. Therefore if the condition fails at first iteration, the enclosed activities are altogether skipped.

switch

BPEL activity `switch` is a multi-way conditional branching control. When there are multiple possible path of execution for a BPEL process, the decision to select a path is taken by `switch` activity. Each of these paths is wrapped in a `case` activity and listed inside `switch`. Just like `while`, each `case` has a condition. The condition for each case is evaluated in the order they are listed under `switch`. If a condition evaluates to True, the corresponding activities specified for that case are executed and the `switch` activity terminates, otherwise the next case condition is evaluated. If all case conditions evaluate to False, the activities specified under optional `otherwise` is executed.

eventHandler

BPEL activity `eventHandler` is used to specify the set of activities to be executed when an event occurs. An event can be either receiving a message or expiration of a timeout. `eventHandler` activity has `onMessage` and `onAlarm` sub-activities to handle these events.

2.5 The Spring Framework

Spring is an open-source, lightweight and loosely coupled Java application framework. It was created by Rod Johnson and described in his book *Expert One-on-One: J2EE Design and Development* [Johnson, 2002]. Although it was created to address the complexity of server side development, it has made its way into a multitude of other Java applications (e.g. Spring Mobile [spr, 2011b] and Spring Android [spr, 2011a]). Furthermore, the tremendous

community support has allowed the framework to be extended even for non-Java applications (e.g. Spring.NET [spr, 2011c]).

The Spring Framework consists of several modules that offer most of the services required to develop enterprise-ready application. As shown in Figure 2.6, these modules are stacked over the core module that forms the basis of Spring framework. It is possible to use a subset of these modules, in combination with core module, to build an enterprise application. Furthermore, Spring offers interfaces to plug-in other frameworks that might be required by an application. Consequently Spring can work in harmony with other frameworks if its modules do not fit the bill.

The core module acts as a container for application objects. It is responsible for creating, configuring and managing these objects. The aspect-oriented programming (AOP) module addresses the system-wide concerns of an application (e.g. security). The data access object (DAO) and object-relational mapping (ORM) modules help in interacting with a data source. The MVC module helps in developing web applications wherein the business logic is segregated from the user interface. This thesis only uses the core module of the framework.

Any non-trivial application (based on object oriented paradigm) constitutes of a set of objects, each corresponding to an instance of a real-world object, concept or data. These objects interact and collaborate to meet the business objectives of the underlying application. Traditionally this interplay required an object to create instances for all objects it would need. As shown below, this is done using *new* in Java programming language.

```
class A {
   B b = new B();
   public int add(int a,int b) {
      return b.sum(a,b);
   }
}
```

```
class B {
   public int sum(int x,int y) {
      return x+y;
   }
}
```

Such a strong coupling among objects make them difficult to test and reuse. This is further exacerbated for enterprise applications that comprises of thousands of objects. The crux of the problem is the necessity for an object to manage its dependencies.

Spring Framework resolves this problem by managing the dependencies among application objects. Instead of objects having to create and maintain their own dependencies, they are injected by the framework. Considering that Spring creates each object in the system, it has all the objects required to resolve any dependency. The process of configuring dependencies among application objects is known as *wiring*. Although Spring supports multiple ways for

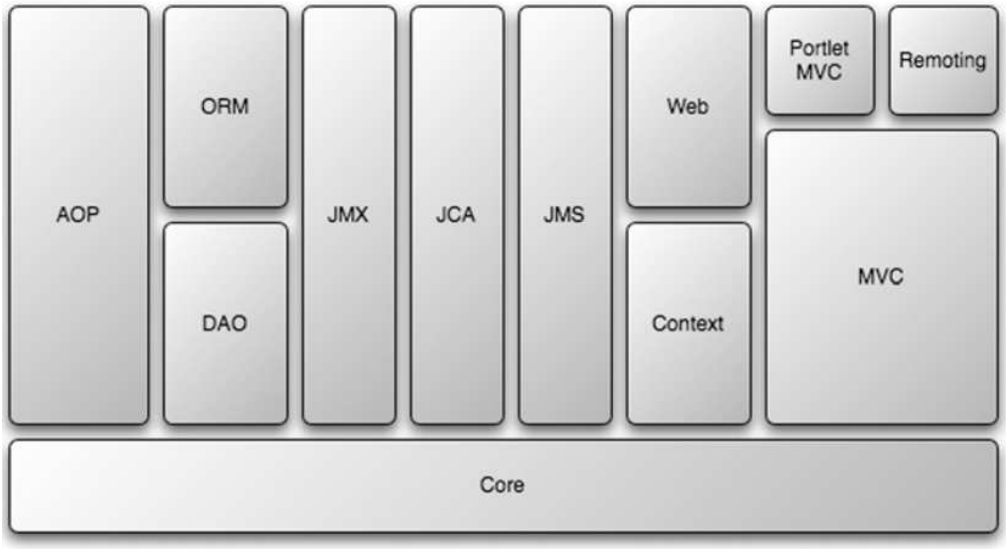


Figure 2.6: The modules of Spring Framework [Walls and Breidenbach, 2007]

wiring components, it is more often done using XML documents. As shown underneath, each application object is instantiated by the framework as a bean. In doing so, it assigns all the dependencies of a bean as its properties. The appropriate *setter* methods are used for wiring.

```
class A {
    B bobj;
    public void setBobj(B bobj) {
       this.bobj=bobj; }
    public int add(int a,int b) {
       return bobj.sum(a,b); }
}
```

```
class B {
    public int sum(int x,int y) {
       return x+y;
    }
}
```

```
< bean id="a" class="A">
    < property name="bobj" ref="b">
< /bean >

< bean id="b" class="B">
< /bean >
```

The core module of the Spring framework acts as the *container* for application objects. It contains an interface `BeanFactory` that creates, configures and manages beans. This interface has several implementations (e.g. `XmlBeanFactory`, `SimpleJndiBeanFactory` etc.), each of

which act as a simple container.

Spring Framework is used in this thesis to formalise a BPEL specification because:

- Our technique for automatic formalisation of a BPEL specification required instantiating and initialising the DTOs corresponding to BPEL activities (as discussed in Chapter 6). The Spring framework offered great convenience in this transformation which resulted in its adoption.
- The Spring framework uses XML based files for specifying beans. This allowed extending the Spring framework to parse the activities in a BPEL specification (which is also XML based) to create their beans.
- Spring helps in developing lightweight and loosely coupled applications.

2.6 JAXB 2 APIs

Java Architecture for XML Binding (JAXB) 2 APIs offer a practical, efficient and standard way of mapping between XML and Java code. While transforming a Java object hierarchy into an XML document is known as *marshalling*, the reverse is known as *unmarshalling*. JAXB 2 APIs support both marshalling and unmarshalling.

It consists of two basic components :

1. The binding compiler (*i.e. xbj*) that transforms an XML schema into a set of Java classes. In doing so, the compiler embeds the structure specified in the XML schema into these classes. This is done using special JAXB annotations that are used by the runtime framework.
2. The runtime framework actually provides the marshalling and unmarshalling features. The annotated classes generated by the compiler is used for this transformation.

An XML schema describes the precise structure of an XML document. This includes the set of elements and the attributes allowed in it. In addition, a schema might contain details about the order and number of child elements allowed for a parent element. When the binding compiler processes a schema, it includes every available informations into the classes produced. The compiler produces a classes corresponding to each element defined in the schema.

```
>xjc test.xsd -p package.test -d src/generated
```

The options p and d are used to specify the package and directory for generated classes.

2.6.1 Unmarshalling an XML document

This requires creating a `javax.xml.bind.JAXBContext` object using the package name for classes that were produced by binding compiler.

```
JAXBContext context = JAXBContext.newInstance("package.test");
```

Unmarshalling a document requires creating a `javax.xml.bind.Unmarshaller` object from the context.

```
Unmarshaller unmarshaller = context.createUnmarshaller();
```

Finally the unmarshaller object processes an XML document to return an object. This object is an instance of the class corresponding to the root element of XML (that was generated by binding compiler). It is worth mentioning that the XML document used herein must adhere to the schema compiled earlier.

```
File f = new File("test.xml");
RootElementClass obj = (RootElementClass) unmarshaller.unmarshal(f);
```

2.6.2 Marshalling Java Object

The process for marshalling is almost the reverse of unmarshalling. As with unmarshalling, initially a `javax.xml.bind.JAXBContext` object is created using the package name for generated classes.

```
JAXBContext context = JAXBContext.newInstance("package.test");
```

Thereafter the class corresponding to root element of envisioned XML documented is instantiated and initialised. The initialisation involves assigning the attributes and sub-elements.

```
RootElementClass obj = new RootElementClass();
obj.setAttribute1 = "att1";
...
```

Finally a marshaller object is created and the object is marshalled.

```
context.createMarshaller().marshal(obj, System.out);
```

The rendered XML is written to standard output. JAXB 2 APIs are used in this thesis because:

- The intermediate DTOs rendered by the Spring framework needs to be marshalled into an XML based formal model.
- JAXB 2 provides a compiler to compile a schema into a hierarchy of classes. These classes help in creating XML documents that confirm to the schema definition.
- They are convenient and powerful.

2.7 Application of the Introduced Technologies

The remainder of the thesis uses the background information as follows:

- CPN semantics are used in Chapter 3 to propose the techniques for reducing the memory costs otherwise involved in model-checking. The reduction is obtained by storing the states as the difference from one of the neighbouring states.
- Hierarchical CPN (or HCPN) semantics are used in Chapter 4 to propose a technique for reducing the time requirements for model-checking. The reduction is obtained by exploring the inter-dependent modules of a HCPN in parallel. These dependencies are stored as parameters in special data-structures. On assigning specific values to these parameters, these dependencies are resolved and the envisioned reachability graph is obtained.
- CPN semantics are used in Chapter 5 to propose a technique for installing hierarchy into a flat model. The technique is based on decrease and conquer technique wherein the bigger problem is broken into smaller problems and the solution to smaller problems are combined to solve the original problem.
- Spring framework is extended in Chapter 6 to automatically transform a BPEL specification into intermediate DTOs. Thereupon JAXB 2 APIs are used to transform the DTOs into an XML based formal model. This solution 1) is extendible for non XML based formal models, and 2) has significantly small transformation time.

Chapter 3

Memory Efficient State-Space Analysis in Software Model-Checking

Formal methods have an unprecedented ability to endorse the correctness of a system. In spite of that, it has been limited to safety-critical and mission-critical systems owing to significant time and memory costs involved. Lately, our ever increasing dependency on software in all walks of life has necessitated using formal methods for a wider range of software. In this chapter, we propose two techniques to reduce the memory requirement for *model-checking*, a widely used formal method. A model-checker stores all explored states in memory to ensure termination. The proposed techniques slash memory costs by storing a state as how different it is from one of its neighbouring states. Our experiments report a memory reduction of 95% with only doubling of computation delay. Aforesaid reduction allows model-checking in a machine with only a fraction of memory needed otherwise. Consequently the advantage is twofold, 1) enormous savings as only a small physical memory is required and 2) as more states can now be stored in a memory of same size, the chances of complete state-space analysis is exceedingly high.

3.1 Motivation

Traditionally, a software is considered "fail-safe" if it has passed a rigorous testing phase [Beizer, 1990]. However, the crash of Ariane 5 launcher [Clarke et al., 2000] and the deaths

due to malfunctioning of Therac-25 radiation therapy machine [Rushby, 1989] in spite of rigorous software testing suggest otherwise. The team investigating these accidents recommended using formal methods (FM) to complement testing as the former assures exhaustive verification of a system [Clarke et al., 2000; Rushby, 1989].

This is further exacerbated by the SOA based applications that are loosely coupled and dynamically composed. Despite the advances in sophisticated techniques to allow automatic matching and composition of web-services [Hao and Zhang, 2007; Paolucci et al., 2002], such applications overwhelmingly rely on automated verification methods to vouch for their credibility and reliability. Considering that model-checking is an automated verification technique that scrutinises all possible behaviours of a system exhaustively, it ought to be used for SOA based applications.

However, considering that most non-trivial systems have gigantic number of states (known as *state space explosion* problem [Christensen et al., 2001]), FM is associated with a high price-tag. This is essentially due to 1) the delay in generating such a large number of states and 2) the enormous space requirements for storing these states. Consequently the developers are often compelled to completely skip FM in order to meet software budget. In this chapter, we propose two techniques for reducing the cost of FM so that they can be more widely used.

Model-checking (MC) [Clarke et al., 2000] is a formal verification method that involves scrutinising the reachable states of a system for a set of predefined undesirable properties. However, during state-space exploration, there might be states generated more than once. To prevent analysing the same states repeatedly for desired properties, it is necessary to remember the states already explored by storing them in memory. This also ensures termination, a condition where no new states are generated. Nevertheless, model-checking is plagued with the state-space explosion problem, wherein a gigantic number of states need to be scrutinised for finding a counterexample. This causes manyfold increase in memory costs, as each new state has to be stored. Such bottlenecks in available memory hinder model-checking.

Some solutions based on 'Partial storage' address the problem by storing only a subset of explored states [Christensen et al., 2001; Mailund and Westergaard, 2004; Godefroid et al., 1993]. Although this reduces the memory requirement, it is difficult to decide the set of states to be deleted (i.e. not stored). If a deleted state is reached again in future, it would be treated as a new state and explored further. The proposed solution has no such issues as it uses 'Exhaustive storage' technique wherein all explored states are compressed and stored in a suitable data structure (e.g. hash-table). The states need to be decompressed before comparison as it is possible for more than one state to have similar compressed state.

In this chapter, we devise two techniques to reduce the memory costs otherwise involved in model-checking. These techniques require storing states as the difference from one of the neighbouring states. Based on the neighbouring state used to calculate the difference, these techniques are classified as : 1) *Sequential model* store a state as how different it is from its immediately preceding state and 2) *Tree model* store a state as how different it is from its nearest state in explicit form. As observed previously, each state produced during state space exploration is remembered by storing it in memory. When the state is stored in any of these alternate forms, there is a significant reduction in memory costs. The reduction is attributed to the assertion that a change in state is smaller than the state itself.

As mentioned previously, the first step in model-checking requires creating a formal representation of the system. This representation depends on the model-checking tool to be used for verification. Some common languages for system representation are PROMELA for SPIN [Holzmann, 2003], C programming language for BLAST [Beyer et al., 2007] and Coloured Petri-Nets (CPN) for CPN Tools [Jensen et al., 2007]. Due to subtle differences between the representation languages, it is difficult to propose a generic memory reduction technique. The models proposed in this chapter specifically target CPN models. However, we do not claim any advantage in using CPN models. The relative storage technique is based on the assertion that "Change in a state is smaller than the state itself" and it will work for all representation languages, as long as this assertion holds. The assertion is valid because systems usually change in many small steps rather than a single large step. Experiments report a 95% reduction in memory, which further endorse our assertion.

Our contributions can be summarised as follows:

1. We propose relative storage techniques to reduce the memory requirement for model-checking by storing states in difference form. The results indicate up to 95% reduction in memory requirement.

2. We propose a backtracking method to transform a difference state into its explicit form. Considering the possibility of multiple explicit states having the same difference state, our backtracking mechanism allows decompressing the states before comparison and thereby prevents false duplicates. Our decompression technique only doubles the time needed to generate the state-space. This is 33% lower than the time taken by [Evangelista and Pradat-Peyre, 2005].

The remainder of the chapter is organised as follows. Section 3.2 introduces the deliberated problem and provides an insight into the tendered solution. Prior to proposing the

Sequential and Tree models in section 3.4, the related works are compiled in section 3.3. We tabulate and plot the experimental results in section 3.5 and discuss the outcome in section 3.6. Finally we summarize our contributions in section 3.7.

3.2 An Overview of the Deliberated Problem & the Proposed Solution

This section discusses the problem in detail and outlines the proposed solution. State-space analysis of a model is done by generating a *reachability graph* [Jensen, 1996].

Definition 1 *The reachability graph of a model C is a directed graph G(C)=(V,E,v_0) wherein*

- *The vertices in V corresponds to the set of reachable states in C.*
- *The edges in E corresponds to the binding elements in C.*
- *The root vertex v_0 corresponds to the initial state of C.*

Each model has a unique initial state and this is represented by the root node of a reachability graph. At its initial state, the system might have a set of enabled events which can bring in a change in state. Each of these events are represented by a separate edge from the root node of the reachability graph and lead to a new node representing the new state. These new states are then analysed for the set of enabled events. For each enabled event, an outgoing edge is added to the corresponding node. This in turn generates another set of new states to be analysed identically. This analysis continues till the set of new states have no enabled event.

However, it might be possible to reach a state from the initial state by executing different sequence of events. Suppose S_0 is the initial state of a model M and let S' be a state reached by the following two sequence of events

$$S_0[e_1\rangle S_1[e_2\rangle S_2[e_3\rangle \cdots\ [e_m\rangle S'$$
$$S_0[e'_1\rangle S'_1[e'_2\rangle S'_2[e'_3\rangle \cdots\ [e'_n\rangle S'$$

where $\forall i\in[1,m]$: e_i and $\forall j\in[1,n]$: e'_j are events and $S_{i-1}[e_i\rangle S_i$ denotes that event e_i in state S_{i-1} leads to state S_i. This is shown in Figure 3.1. If $\exists i\in[1,n]:(i<m) \wedge (e_i \not\equiv e'_i)$, the state S' can be reached using two different sequence of events and therefore it is a *duplicate state*. The reachability graph for M will have two nodes representing the same state S'. However, analysing both the nodes and their children (each of which will also have a duplicate node) is a waste of resources. The larger the number of duplicate nodes for a state, the greater the wastage in

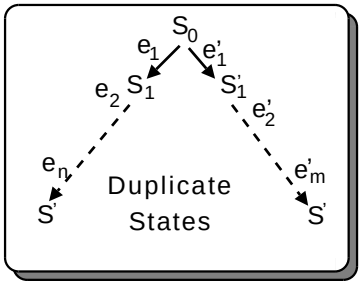


Figure 3.1: S' reached using two different sequential of events from S_0

resources. Furthermore, if there exists a non-empty sequence of events $[e_1 e_2 \cdots e_r\rangle$:r>0 that cause no net change in state of a model M, the model checker might never terminate. This is shown in Figure 3.2. Let S be some state of model M and $\forall i \in [1,r]$: e_i be events such that

$$S[e_1\rangle S_1[e_2\rangle S_2[e_3\rangle \cdots \ [e_r\rangle S$$
$$\text{or } S[e_1 e_2 e_3 \cdots e_r\rangle S$$

Such state of affairs would lead to analysis of the set of states $\{S, S_1, S_2, \cdots, S_{r-1}\}$ forever and state-space analysis might never finish. Consequently, it is necessary to remember the states

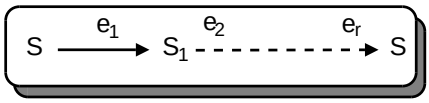


Figure 3.2: The sequence of events $[e_1 e_2 \cdots e_r\rangle$ causes no net change in state

already explored and ignore any duplicate states encountered. A model-checker remembers explored states by storing them in memory. When a state is generated during state-space exploration, it is compared with the stored states to determine if it is new or duplicate of a previously generated state. If it is a duplicate state, the corresponding node in the reachability graph becomes a terminal node and it is not analysed any further. Otherwise, the new state is stored in memory and is analysed for enabled events. However, due to state-space explosion, large amount of memory is needed to store all unique states. In this chapter, we propose two relative storage techniques to reduce the memory requirement by storing a state as how different it is from its previous state.

At any time, there might be thousands of explored states stored in memory. Comparing each state generated with all stored states might take long. Hence the states are stored in a hash-table to ensure constant time lookup.

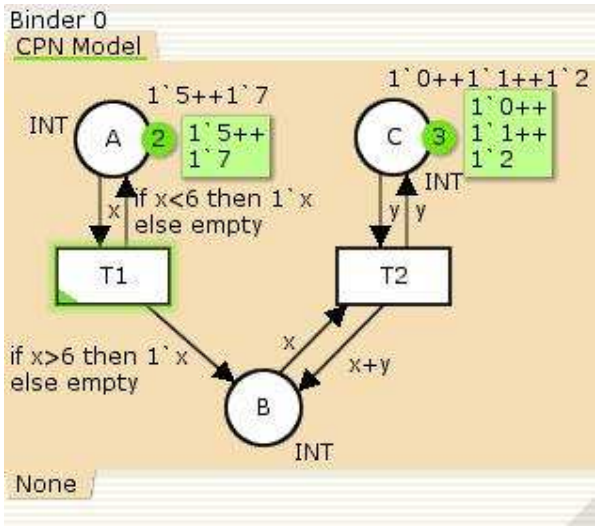


Figure 3.3: A Coloured Petri-Net model. Variables x and y are of type INT

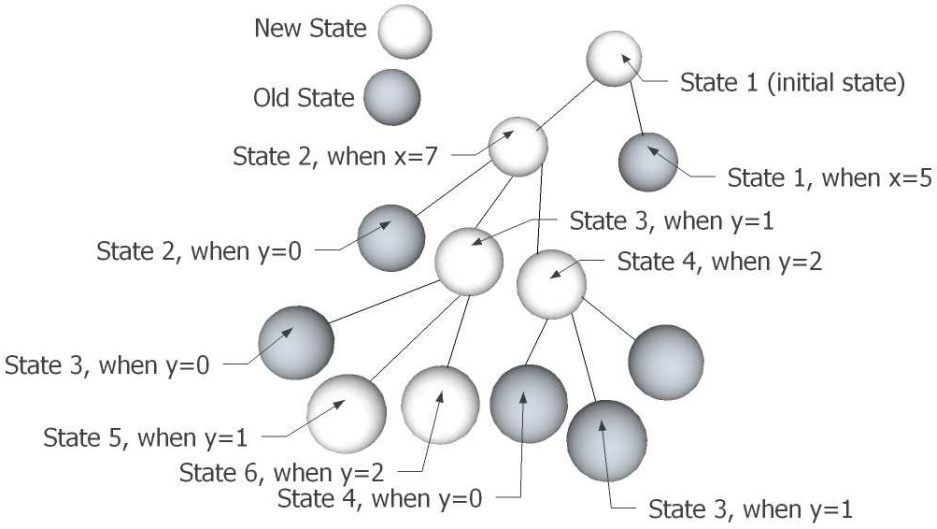


Figure 3.4: A part of reachability graph for CPN model in Figure 3.3

We illustrate the problem using an example. Figures 3.3 and 3.4 show a Coloured Petri-Net model and a part of its reachability graph. All duplicate nodes in Figure 3.4 are shaded. Initially, the CPN model has 2 tokens in place A and 3 in C. This is represented by State 1 in Figure 3.4 and being the root node of reachability graph, it is stored in memory. The enabled events at this state are (`T1`,`x=5`) and (`T1`,`x=7`), where `T1` is the enabled transition and `x`=5 or 7 is the binding for which it is enabled. Corresponding to these two enabled events, the root node in Figure 3.4 has two outgoing edges, one for each event. When `T1` fires with `x`=5, there is no change in state as all tokens remain in their previous places and the edge corresponding to this event leads to a duplicate state in Figure 3.4. Duplicate states are not analysed any further to save time and to ensure termination. The other event (`T1`,`x=7`) results in moving a token from A to B, leading to State 2. Being a new state, it is represented using a bright node in Figure 3.4. Furthermore, it is stored and further analysed for enabled transitions. State 2 has three enabled events: (`T2`,`y=0`), (`T2`,`y=1`) and (`T2`,`y=2`). The first event causes no change in state and is represented by a shaded node in Figure 3.4. The other two events change the value of token in B leading to State 3 and State 4 and these are represented by bright nodes in Figure 3.4. Being new states, they are stored in memory and further analysed for enabled events. Remaining states are explored analogously to generate the complete reachability graph.

The reachability graph in this example has an infinite number of states. Other models might have finite number of states. However, the number of states is almost always gigantic, leading to state-space explosion [Clarke and Berezin, 1998]. Complete state-space analysis is possible only when the available memory (α_A) in a machine is at least equal to the memory needed to store all unique states in the reachability graph (α_M) of model M. Otherwise, if $\alpha_A < \alpha_M$, only a partial state-space analysis can be performed and the analysis stops when memory is full. This chapter describe models for compact representation and storage of a state so as to dramatically reduce the memory requirements for model-checking. Using the proposed models, the memory needed to store all unique states in reachability graph of a model M reduces to α'_M. This allows 1) Complete reachability analysis in a machine with a smaller memory α'_A if $\alpha'_A \geq \alpha'_M$, where $\alpha'_A < \alpha_A$ and $\alpha'_M < \alpha_M$ 2) Even when $\alpha_A < \alpha'_M$, the partial state-space analysis can have at least a few more steps. With available memory remaining same, we are able to create a reachability graph with more states due to less memory needed to store a state.

3.3 Related Work

All solutions proposed to store state-space can be classified as either of 1) Partial storage 2) Lossy storage or 3) Exhaustive storage. The proposed solution is based on exhaustive storage.

As the name indicates, only a subset of the explored states are stored in *Partial storage* techniques. Sweep-line method [Christensen et al., 2001] is such an approach, wherein a state is deleted if it cannot be reached again in future. However, it is difficult to decide the states to be deleted. Furthermore, it is not a generic solution as for different systems, we might need to delete a different set of states.

Lossy storage techniques produce the entire state-space wherein each explored state is compressed and stored in a suitable data structure (e.g. hash-table) to ensure a constant time lookup. However, the compression algorithm used is not reversible. In order to determine if a state is new, it is also compressed and compared with stored states. As pointed out previously, multiple states can have the same compressed form. This often results in falsely identifying a state as duplicate. Some interesting solutions based on lossy storage are:

1. *Ordered Binary Decision Diagram (OBDD) with compression* uses decision tree to store

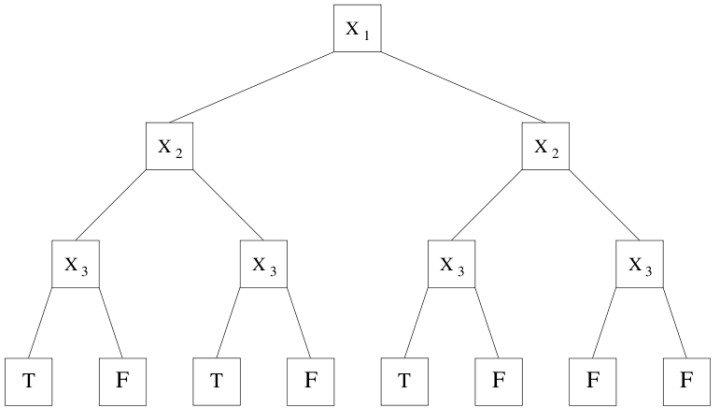


Figure 3.5: Decision tree for boolean function $f(x_1, x_2, x_3) = \overline{x_1}.\overline{x_2}.\overline{x_3} + \overline{x_1}.x_2.\overline{x_3} + x_1.\overline{x_2}.\overline{x_3}$

visited state [Visser, 1996]. Figure 3.5 shows an acyclic graph that can store 8 possible states. These states are represented using 3 bits, from 000 to 111. Each left arm of the acyclic graph represents 0 while each right arm represents 1. For instance, starting from the root, a state 100 would use the right arm at the first level and the left arm at the remaining two levels. Finally, after traversing all the arms based on bit-values, a

terminal node is encountered that stores either *true* or *false*. A terminal node with *true* denotes that the marking which leads to it has been visited. Consequently the terminal nodes for an acyclic graph need to be updated whenever a new state is generated.

The compression algorithm divides the bits representing a state into p+1 parts such that part d has n_d bits. Each of these bit pieces have a table and the table for d^{th} bit piece have k_d entries such that m_d=$\log_2 k_d << n_d$. This reduces the size of graph by ensuring that a state is represented using less number of bits. Figure 3.6 illustrates the compression technique.

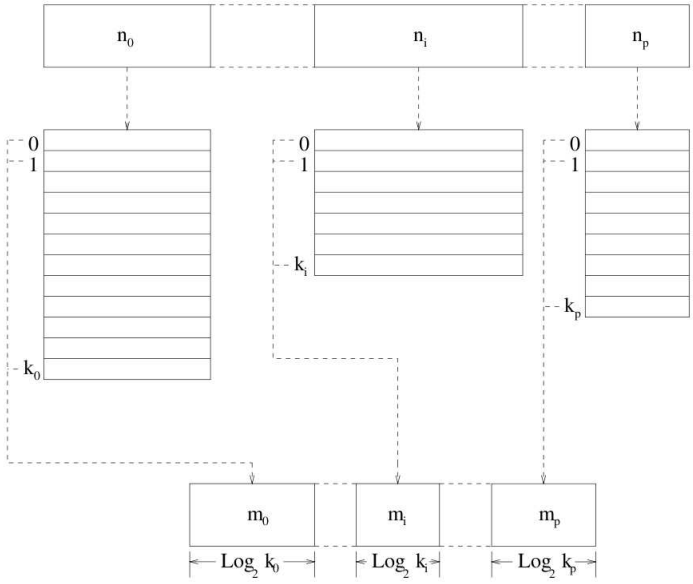


Figure 3.6: Compression of states represented by bits

2. *Automaton representation of reachable states* store states as a sequence of bits [Holzmann and Puri, 1999]. This is similar to OBDD where x bits were used to represent 2^x states. However, instead of an acyclic graph, an automata is used to store the states. Its edges are inscribed with bit values bit values *0* & *1*. Starting from the root node, the edges are followed based on the bits representing a state. If the terminal node has a *1*, the state is established to be a duplicate. Figure 3.7 shows an automata that leads to a *1* node for the set of states represented by {000, 001, 101}. The automata is modified whenever a new marking is generated to ensure that its corresponding bit sequence leads to a *1* node.

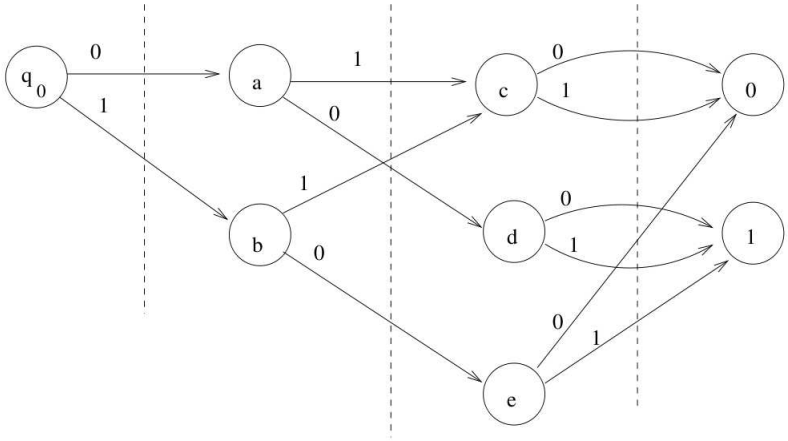


Figure 3.7: Storing visited states using automata

3. *Graph encoded tuple sets (GETS)* allows a compact representation of states wherein the common prefix and suffix for a set of states are used to reduce the size of graphs. The reduction is obtained only when the state space is large. Otherwise there might be an expansion of required storage instead of the envisioned reduction. Results indicate that GETS can decrease the memory requirements up to 7 times with a tripling of processing time.

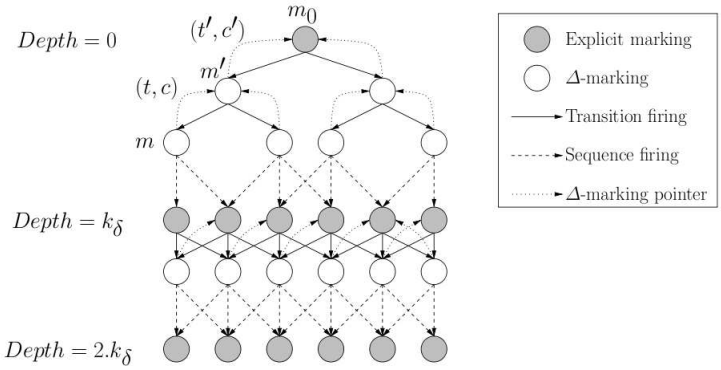


Figure 3.8: Generating state-space with △-mappings [Evangelista and Pradat-Peyre, 2005]

Exhaustive storage techniques also produce the entire state-space wherein each explored state is compressed and stored in a suitable data structure (e.g. hash-table) to ensure a constant time lookup. However, unlike lossy-storage, the compression algorithm needs to be essentially reversible as otherwise the states cannot be regenerated for comparison. Both the

Tree and Sequential models incorporate compression techniques that are reversible. Other compression techniques include $\triangle$-markings [Evangelista and Pradat-Peyre, 2005] shown in Figure 3.8 wherein a state is stored as the transition instance that leads to it from the previous state. Such solutions are specific to Petri-Nets and other related formalisms where the transitions are deterministic. Nevertheless, the compression rendered has an inevitable associated delay. The existing models offer a generic solution within an acceptable time-frame.

Table 3.1: A comparison of solutions based on exhaustive storage

Method	**Run-Time**	**Memory-Use**
No Algorithm	100%	100%
[Schmidt, 2003]	130%	60%
[Evangelista and Pradat-Peyre, 2005]	300%	05%
[Holzmann, 1997]	280%	18.3%
Sequential Model	200%	05%

Table 3.1 compares the sequential model with other solutions based on this approach and the state-space compression they provide. The table also gives the additional delay incurred when using a solution. The sequential model provides reduction equivalent to [Evangelista and Pradat-Peyre, 2005] with only 2/3 of its delay.

Table 3.2: A comparison of the storage techniques for memory-reduction

Criteria	**Storage Techniques**		
	Partial Storage	*Lossy Storage*	*Exhaustive Storage*
States Stored	◐	●	●
Compression Algorithm	$\otimes$	$\not\rightleftarrows$	$\rightleftarrows$
False Positive	×	✓	×
False Negative	✓	×	×
Generic	×	✓	✓

Table 3.2 illustrates the advantage of exhaustive storage over other related techniques. In case of *false positive*, two states are erroneously implicated identical. Similarly the failure in identifying duplicate states is termed as *false negative*. The proposed models are based on exhaustive storage techniques that are unstained with aforementioned shortcomings.

3.4 Proposed Models for Memory Efficient State-Space Analysis

In this section, the Sequential & Tree models are proposed for memory efficient state-space analysis. Although these models specifically target CPN models, we do not claim of any advantage in using them. As stated previously, the relative storage technique is based on the assertion that "Change in a state is smaller than the state itself" and as long as this assertion holds, it is valid for all modeling languages. For CPN, a change in state occurs if one or more tokens (1) move to another place, or (2) change their value, or (3) are created in the model, or (4) are deleted from the model or (5) combination of these such that the colour of each token match the colour-set of containing place. In a CPN model, these changes are brought in by a transition. However, a transition usually modifies the place and value information of only a small number of tokens. Furthermore, a very small number of tokens are usually created or deleted by a transition. Therefore it is substantially cheaper to store how different a state S is from its previous state than storing the full state S. Based on this, we propose the Sequential model to generate a memory efficient reachability graph in next section.

3.4.1 The Sequential Model

The Sequential model aims to reduce the memory requirements for model-checking. Such a reduction will increase the affordability and consequent use of model-checking in software development. The focus of this model is storing states in difference form, which is defined as follows:

Definition 2 *The difference form of a state S_{st}, denoted as D_{st}, is defined as the changes necessary in its previous state S_{pv} to generate it.*

Corollary 1 *If D_{st} is the difference form of a state and S_{pv} is its previous state, the state S_{st} can be regenerated in explicit form as S_{st}=S_{pv}+D_{st}.*

An explicit state has information for all tokens, and is often referred to as state in this chapter by omitting the adjective 'explicit'. Below we illustrate how to find the difference form of a state using a simple example.

Figure 3.9 presents a portion of reachability graph for the CPN model in Figure 3.3 using sequential model. Initially the model is in State 1 and since it does not have a previous state, it is stored in explicit form in Figure 3.9. The tokens in the model are arbitrarily named a to e in Figure 3.9. Furthermore, their places are assigned by an arrow ($\rightarrow$) and values are

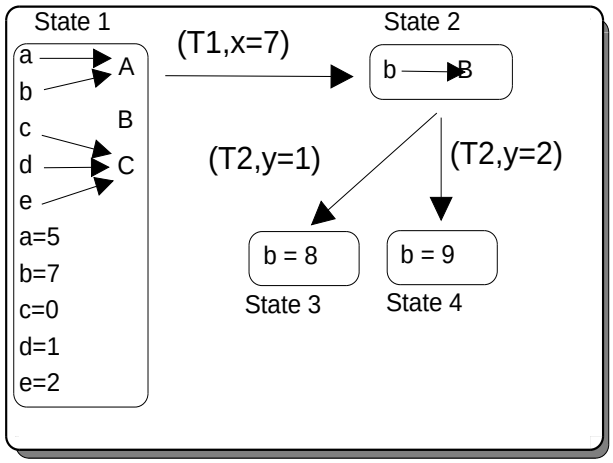


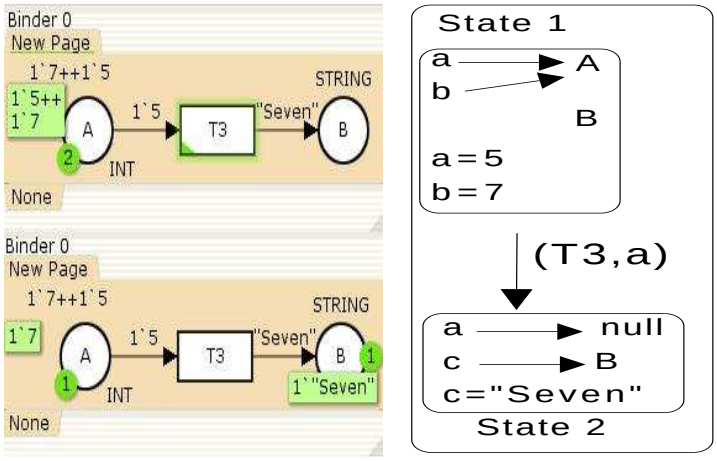
Figure 3.9: A part of reachability graph in Figure 3.4 using sequential model. The corresponding model is shown in 3.3

assigned by an equal (=) symbol. For example, the token 1'5 in place A of Figure 3.3 is named `a`. Its place is assigned as `a`→A while the value is assigned as `a`=5. Similarly, token 1'7 in A is named `b` and assigned place and value as `b`→A and `b`=7.

When the event (`T1,x=5`) occurs, the model persists in State 1. As a result, the difference form is empty (or null) and not drawn in Figure 3.9. However, (`T1,x=7`) takes it to State 2. In order to store the new state in difference form, we need to find the changes in State 1 brought by this event. We find that the event moved token 1'7 to place B. This information is sufficient to construct State 2 from State 1. We therefore store State 2 in difference form as `b`→B.

The event (`T2,y=1`) in State 2 leads to State 3. Likewise, the event (`T2,y=2`) leads to State 4. In order to store these new states in difference form, the changes in State 2 brought by these events need to be found. On inspecting these events, both are found to change the value of token in place B. While (`T2,y=1`) changes the value to 8, (`T2,y=9`) changes it to 9. Given State 2 in explicit form, this information is sufficient to construct State 3 and State 4. Accordingly, State 3 is stored as `b`=8 while State 4 is stored as `b`=9. The difference form for other states are calculated identically. Additionally, each state also store a pointer to its previous state. This is necessary to regenerate the states as explained later. As evident from this example, it takes less space to store states in difference form.

A state change also occurs when an event creates or deletes one or more tokens. If an event deletes a token `a`, the new state can be represented in difference form by assigning the place for `a` as `null`(`a`→`null`). Similarly when an event creates a new token, it is given an arbitrary



(a) A CPN model where T3 fires to delete token 1'5 and create token 1'"Seven"

(b) Reachability graph using sequential model

Figure 3.10: State change when tokens are created and/or deleted

name and assigned the corresponding place and value information. This is illustrated by an example in Figure 3.10. The CPN model in Figure 3.10(a) has a transition `T3` which removes token `1‘5` from place A and adds `1‘"Seven"` to place B. Considering the value in latter token, place B is assigned colour-set STRING. The reachability graph of the model using sequential model is presented in Figure 3.10(b). The tokens in place A are assigned names `a` and `b`. The initial state of the model is stored explicitly in Figure 3.10(b) as there is no previous state to calculate difference. When event (`T3,1’5`) occurs, it deletes the token `a` and creates a new token which we name `c`. The new state can be stored in difference form by assigning the place of `a` to `null` and assigning the place and value information for newly created token. Given State 1 in explicit form, aforesaid information is sufficient to regenerate State 2. This example further endorse a reduction in memory requirement when states are stored in difference form.

In this section, we explained how to obtain the difference form of a state and demonstrated the memory reduction when states are stored in difference form. However, more than one explicit state can produce the same difference state. This necessitates converting states into explicit form before comparison. This is explained with an example in next section.

A)Expanding a State in Difference Form

Here we demonstrate backtracking in order to revert a difference state. This is necessary for comparison as more than one explicit state can produce the same difference state.

When a state is generated during state-space exploration, it has to be compared with stored states to determine if it is new. However, compressing and comparing it with states stored in difference form might lead to an error owing to multiple states having the same difference form. Supposing three states S_a, S_b and S_c produce the same difference form D_{abc}. When either of the three states is encountered for the first time, D_{abc} is stored in memory. When the other two states are encountered and compared with stored states in compressed form, they are wrongly interpreted as duplicate state. Therefore it is essential to revert a stored state before comparing. Such a conversion is called *expanding* and is done by *backtracking.*

Definition 3 *Backtracking is the process of regenerating a state by recursively adding the most recent changes for each token to its previous state until a state stored in explicit form is reached.*

Corollary 2 *If S_n and D_n are the explicit and difference states at depth n of a reachability graph, the former can be obtained from latter using a backtracking function BK, where S_n=$BK(D_n)$=D_n+S_{n-1}.*

Corollary 3 *Since S_0 is always in explicit form, the equation in Corollary 2 can be solved for all $n \leq h$, where h is the height of the reachability graph.*

We illustrate backtracking with an example. State 4 is stored in difference form in Figure 3.9 and in order to expand it, we need to backtrack till an explicit state is encountered, as shown in Figure 3.11. Initially, State 4 contains new value for token `b` and a link to its previous state. Using Definition 2, we should get State 4 in explicit form by updating its previous state (which we hope is in explicit form) with this value. However, on backtracking one step in Figure 3.9, we reach State 2 which is also stored in difference form. But it gives additional information about the place of token `b` and a link to its previous state. We add the place information from State 2 with value information from State 4 and get a meta-state 4(1) shown in Figure 3.11. We call 4(1) a meta-state as it was obtained by combining two different states. Using Definition 3, State 4 can now be obtained by updating the previous state of State 2 with information in metastate 4(1). On further backtracking, State 1 is

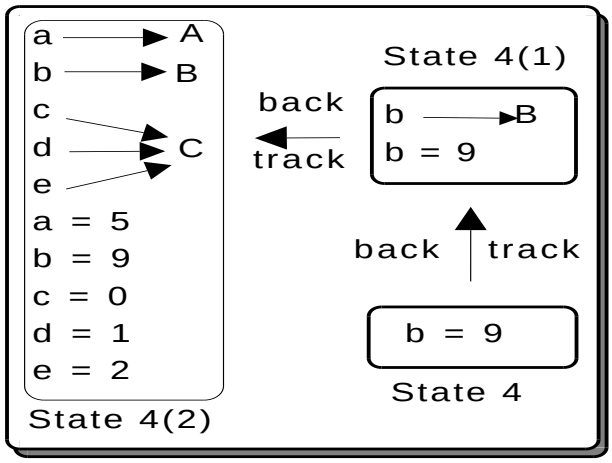


Figure 3.11: Backtracking to expand State 4 in Figure 3.9

encountered which is actually in expanded form. We update it with the information in 4(1) and get another metastate 4(2). By Definition 3, 4(2) is State 4 in expanded form.

As backtracking is an additional overhead when states are stored in difference form, they increase the time needed for state-space analysis. Definition 3 requires backtracking to initial state S_0 for expanding each state. However, this leads to large delay with increase in height of reachability graph. In the next section, we discuss ways for reducing this overhead.

B)Decreasing the Cost of Expanding

Here we discuss different ways of reducing the additional delay incurred while backtracking. Reducing this delay will reduce the overall time for model-checking.

So far we have stored only the initial state in explicit form while all other states to be stored in difference form. Although this should ensure maximum reduction in memory requirements, Definition 3 will require backtracking to the initial state for expanding. As a result, the states far from initial state take long to expand.

In order to reduce the delay, the number of backtracking steps need to be minimised. If every state at depth $i*\delta$: $i \in N$[1] steps from initial state is stored in expanded form, expanding a state will never need a backtracking greater than δ-1 steps. Therefore starting from initial state, δ^{th}, $2\delta^{th}$, $3\delta^{th} \cdots$ states are stored in expanded form. The model can be tuned by accepting different values of δ. Finally, we propose the algorithm for Sequential model in next section.

[1]N is the set of natural numbers starting from 0

C)The algorithm for Sequential Model

The algorithm for Sequential Model is introduced here. As specified previously, sequential model achieves memory reduction by storing states in difference form. Starting from the initial state, it explores remaining reachable states of the model using depth-first search (DFS) algorithm [Cormen et al., 2001]. Each explored state is stored in a hash-table.

When a state is generated, a hash-function is used to find its index in a hash table. If this index is empty, the state is new. Its difference form is calculated and inserted at this index. Furthermore, the enabled transitions are identified and one of them is fired. Otherwise, if there are states already stored at this index, they are all expanded and compared with the generated state. If there is no match, the state is new and it is inserted at the head of list at this index. Additionally, one of its enabled transitions is executed. In case of a match, the state is duplicate of a state analysed previously. It is neither stored nor analysed for enabled transitions.

The proposed algorithm calculates the difference form of a state by comparing it with its previous state. However to reduce delay in backtracking, all states at depth i$*\delta$: i$\in$N from initial state are stored in explicit form, where δ is the shortest distance between two explicit states. In order to expand a state, the algorithm implements backtracking until an explicit state is encountered.

The proposed algorithm also implements two-level hashing at an index if the number of states stored at that index exceeds a threshold. In our algorithm, we set threshold as M/10, where M is an estimation of the total number of reachable states. When two level hashing is used, the index of primary hash table contains hash-function for secondary hash table.

The proposed algorithm for sequential model has three parts:

1. `SEARCH:` The steps are listed in Algorithm 1. This algorithm accepts a state (S_{st}), it's previous state (S_{pv}) and the distance of S_{st} from last expanded state (depth) as input. A hash function H is used to find the index for state S_{st} as shown in step 1. The algorithm then checks the content of hash-table at this index. There can be three possibilities.

   (a) *Hash table contain NULL at this index:* In this case, it is the first time this state is generated. Hence Algorithm 2 is called to store the state at this index. Any enabled event at this state is fired. Steps 2-4 in Algorithm 1 check and handle this case.

(b) *Hash table contain a linked-list at this index:* In this case, each state in the linked-list has hashed to this index. S_{st} is compared with each state in this list. If a state is stored in difference form, it is expanded before comparison using Algorithm 3. In case of a match, the state is neither stored nor analysed for an enabled event. Otherwise, the state is stored at the head of linked list using Algorithm 2 and an enabled event is fired. Steps 5-11 in Algorithm 1 check and handle this case.

(c) *Hash table contain a hash-function at this index:* If this case, all states which hashed to this index are stored in a separate hash-table HASHi indexed by the function H' stored at this index. In step 14, the index in second hash table is calculated using this hash function. Step 15 checks if this index is empty or has a state stored. In case this index is empty or does not contain this state, it is inserted at this index using Algorithm 2 and its enabled events are fired. Otherwise the algorithm returns. Steps 15-24 in Algorithm 1 handle these cases.

2. `INSERT:` This algorithm is responsible for inserting a state into hash table and is listed in Algorithm 2. It accepts a state (S_{st}), it's previous state (S_{pv}) and the distance of S_{st} from last expanded state (depth) as input. The fields of a pointer "new" are assigned the required values before storing it in appropriate index. Based on the value of delta, the state is either stored in explicit or difference form and this is assigned to 'type' field of pointer 'new'. In case of former, the explicit state S_{st} is assigned to 'state' field of 'new'. Otherwise the difference, given by "S_{st}-S_{pv}" is assigned to 'state' field. Additionally, in latter case, a pointer to previous state is stored in 'prev' field of 'new'. This is shown in steps 1-8 of Algorithm 2.

   The index of hash-table at which this state is to be stored is calculated in step 9. There could be three possible cases:

   (a) *Hash table contain NULL at this index:* This is the case when S_{pv} is generated for the first time. The contents of pointer 'new' is simply copied to this index of hash-table. This is shown in steps 10-11 of Algorithm 2.

   (b) *Hash table contain a linked-list at this index:* In this case, the contents of 'new' is copied to head of linked list. Furthermore, it is checked if the list contains more than 10% of an estimated total number of states. In that case, the states in this linked list is stored in another hash table and the hash function is stored at this index. This is done in steps 12-22 of Algorithm 2.

Algorithm 1: SEARCH (State S_{st}, int depth, State S_{pv})

Data: current state S_{st},steps away from last explicit state (depth), previous state S_{pv}
Result: Decide if a state generated is new

```
1  i←H[S_st] ;
2  if HASH[i]=NULL then
3  |  INSERT(S_st,depth,S_pv);
4  |  foreach S' such that S_st[(t,c)⟩S' do SEARCH(S',(depth+1)mod δ,S_st);
5  else if HASH[i] points to a linked list then
6  |  foreach state D in linked list do
7  |  |  if D is in difference form then D←RECONSTRUCT(D);
8  |  |  if D=S_st then return;
9  |  end
10 |  INSERT(S_st,depth,S_pv);
11 |  foreach S' such that S_st[(t,c)⟩S' do SEARCH(S',(depth+1)mod δ,S_st);
12 else if HASH[i] contains a hash function then
13 |  H' ← HASH[i];
14 |  j←H'[S_st] ;
15 |  if HASHi [j] is empty then
16 |  |  INSERT(S_st,depth,S_pv);
17 |  |  foreach S' such that S_st[(t,c)⟩S' do SEARCH(S',(depth+1)mod δ,S_st);
18 |  else
19 |  |  if HASHi[j] is in difference form then
20 |  |  |  HASHi[j]←RECONSTRUCT(HASHi[j]);
21 |  |  end
22 |  |  if HASHi[j]=S_st then return;
23 |  |  INSERT(S_st,depth,S_pv);
24 |  |  foreach S' such that S_st[(t,c)⟩S' do SEARCH(S',(depth+1)mod δ,S_st);
25 |  end
26 end
```

(c) *Hash table contain a hash-function at this index:* In this case, the hash-function stored at this index is used to find the index in secondary hash-table and the contents of pointer 'new' is copied to that index. Steps 23-26 in Algorithm 2 handle this case.

3. `RECONSTRUCT`: This algorithm accepts a difference state and expands it by backtracking. The steps are listed in Algorithm 3. In steps 1-4, the algorithm backtracks and add each state encountered until an explicit state is reached. Finally, the state D_{st} in explicit form can be calculated by adding the sum to the explicit state encountered. This is

Algorithm 2: INSERT(State S_{st}, int depth, State S_{pv})

Data: current state S_{st},steps away from last explicit state(depth), previous state S_{pv}
Result: Insert state S_{st} into hash table

```
1  if depth=0 then //when depth is 0
2  |  new.type←explicit;                          /* store in explicit form */
3  |  new.state←S_st;
4  else
5  |  new.type←difference;                        /* else difference form */
6  |  new.state←S_st-S_pv;
7  |  new.prev←S_pv;
8  end
9  i← H[S_st] ;
10 if HASH[i]=NULL then
11 |  HASH[i]=new;                                /* No conflict */
12 else if HASH[i] points to a linked list then
13 |  insert new at the head of linked list;  /* Multiple states hash to same index
   |  */
14 |  if length(linked list)≥ |M|/10 then
15 |  |  foreach state d in linked list do
16 |  |  |  if d is in difference form then
17 |  |  |  |  d←RECONSTRUCT(d);                /* get explicit state */
18 |  |  |  end
19 |  |  |  add d to HASHi[H'[d]];          /* store in secondary hash table */
20 |  |  end
21 |  |  HASH[i]←H';                          /* store secondary hash function */
22 |  end
23 else if HASH[i] points to a hash function then
24 |  H' ← HASH[i];                           /* get secondary hash function */
25 |  j←H'[S_st] ;
26 |  HASHi[j]←new;
27 end
```

shown in step 5.

Complexity Analysis

The proposed algorithm for Sequential model reduces the amount of space necessary to store the states by using difference states. However, this reduction process is accompanied by a delay due to backtracking. In this section, we calculate the reduction provided and derive the time needed for extra processing.

Algorithm 3: RECONSTRUCT(State D_{st})

Data: State D_{st} in difference form
Result: Expanded form of D is returned

```
1 while d.type=difference do
2 |   sum=sum+d.state ;
3 |   d=d.prev;
4 end
5 return d+sum;
```

Let δ be the distance between two expanded states. We pointed out earlier that the initial state is stored in expanded form. Other states in expanded form are those at depth δ, 2δ, and so on. If a reachability graph has height n, the depth of last expanded node is $\lfloor \frac{n}{\delta} \rfloor * \delta$.

Assuming that the average number of new states generated by a transition is k(>1), the number of states at depth d is given by k^d. This is illustrated in Figure 3.12. All dark circles represent explicit states while shaded circles represent difference states. The number of expanded states in a reachability graph of height n is the sum of the number of expanded states at depth 0, δ, 2δ, $\cdots$, $\lfloor \frac{n}{\delta} \rfloor * \delta$. This is given by

$$\beta_{expanded} = k^0 + k^{\delta} + k^{2\delta} + \cdots + k^{\lfloor \frac{n}{\delta} \rfloor * \delta}$$

This is a geometric progression [Bronshtein et al., 1997] with initial term a=1 and ratio r=k^{δ}. Hence, the sum is given by

$$\beta_{expanded} = \frac{a(r^{n+1} - 1)}{r - 1} = \frac{k^{(\lfloor \frac{n}{\delta} \rfloor + 1) * \delta} - 1}{k^{\delta} - 1} \tag{3.1}$$

Similarly, the total number of states is another geometric progression with initial term a=1 and ratio r=k.

$$\beta_{total} = k^0 + k^1 + k^2 + \cdots + k^n = \frac{a(r^{n+1} - 1)}{r - 1} = \frac{k^{n+1} - 1}{k - 1} \tag{3.2}$$

Therefore the number of states in difference form is given by

$$\beta_{difference} = \beta_{total} \text{ - } \beta_{expanded}$$

Assigning $\beta_{expanded}$ from equation 3.1 and β_{total} from equation 3.2 we get

$$\beta_{difference} = \frac{k^{n+1} - 1}{k - 1} - \frac{k^{(\lfloor \frac{n}{\delta} \rfloor + 1) * \delta} - 1}{k^{\delta} - 1} \tag{3.3}$$

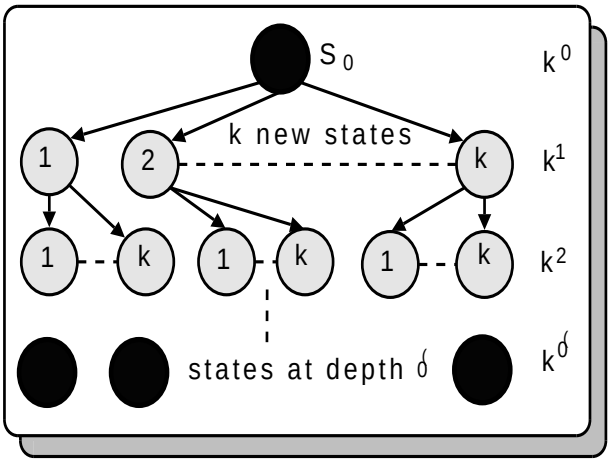


Figure 3.12: At depth d, the number of states is k^d. All states at depth δ are explicit

Percentage Reduction in Memory: The number of states in difference and explicit forms are given by equations 3.3 and 3.1. Suppose the memory occupied by an explicit state is λ, while a state stored in difference form occupies x$*\lambda$ memory, where 0<x<1. Therefore the memory needed to generate a reachability graph of depth n without using our algorithm is

$$\Lambda_{withoutalgo} = \beta_{total} * \lambda \tag{3.4}$$

When using our algorithm, the memory needed to generate the same reachability graph is

$$\Lambda_{withalgo} = \beta_{difference} * \lambda * x + \beta_{expanded} * \lambda \tag{3.5}$$

The percentage reduction in memory denoted by Δ is

$$\Delta = \frac{\Lambda_{withoutalgo} - \Lambda_{withalgo}}{\Lambda_{withoutalgo}} \tag{3.6}$$

Using equations 3.4 and 3.5 in 3.6 we get

$$\text{or } \Delta = \frac{\beta_{total}*\lambda - \beta_{difference}*\lambda*x - \beta_{expanded}*\lambda}{\beta_{total}*\lambda}$$

Substituting $\beta_{difference}$ as $\beta_{total} - \beta_{expanded}$

$$\Delta = \text{(1-x)}*\left(1 - \frac{\beta_{expanded}}{\beta_{total}}\right)$$
$$\text{or } \Delta = \text{(1-x)}*\left(1 - \frac{(k^{(\lfloor \frac{n}{\delta} \rfloor + 1)*\delta} - 1)*(k-1)}{(k^{\delta}-1)*(k^{n+1}-1)}\right)$$

Time needed for Extra Processing: The extra time required when states are stored in difference form is now calculated. We only consider the delays due to backtracking as any other delay is common for both explicit and difference states.

Let i be an integer between 1 and n. If the height of reachability graph is i, the number of states in explicit and difference forms are given by

$$\beta'_{total} = \frac{k^{i+1}-1}{k-1},\ \beta'_{expanded} = \frac{k^{(\lfloor \frac{i}{\delta} \rfloor+1)*\delta}-1}{k^{\delta}-1}$$
$$\text{and } \beta'_{difference} = \frac{k^{i+1}-1}{k-1} \text{ - } \frac{k^{(\lfloor \frac{i}{\delta} \rfloor+1)*\delta}-1}{k^{\delta}-1}$$

When a state S_{st} is generated, it is compared with the state stored at an index given by the hash-function. The probability that this state is stored in expanded or difference form can be calculated as

$$P_{expanded} = \frac{\beta'_{expanded}}{\beta'_{total}} \text{ and } P_{difference} = \frac{\beta'_{difference}}{\beta'_{total}}$$

If the state is stored in difference form, it has to be first expanded by backtracking and then compared with S_{st}. Hence, time taken for comparing S_{st} with the stored states is given by

$$T_{comparison} = T_{expanded} + T_{difference}$$

Suppose the time for comparing two expanded states is ϵ, while it takes y$*\epsilon$ time for backtracking a single step. In the worst case, a backtracking of $(\delta - 1)$ steps is necessary to expand the state. Therefore the time can be calculated as

$$\begin{aligned} T_{comparison} &= P_{expanded} * \epsilon + P_{difference} * \epsilon(\delta - 1)y \\ &= \frac{\epsilon(1-y(\delta-1))\beta'_{expand}}{\beta'_{total}} + \epsilon(\delta - 1)y \\ &= \frac{\epsilon(k^{(\lfloor \frac{i}{\delta} \rfloor+1)*\delta}-1)(k-1)(1-y(\delta-1))}{(k^{\delta}-1)(k^{i+1}-1)} + \epsilon(\delta - 1)y \end{aligned}$$

This is the time taken for comparing a state generated with a stored state in difference form. All comparison at a particular depth takes place concurrently. Hence the total time taken to generate a reachability graph of height n is the sum of time taken for one comparison at each level. This is denoted by π, where

$$\pi = \sum_{i=0}^{n} \frac{\epsilon(k^{(\lfloor \frac{i}{\delta} \rfloor+1)*\delta}-1)(k-1)(1-y(\delta-1))}{(k^{\delta}-1)(k^{i+1}-1)} + \epsilon(\delta - 1)y$$

Since $\lfloor \frac{n}{\delta} \rfloor$=0 for 0≤i< δ, $\lfloor \frac{n}{\delta} \rfloor$=1 for δ ≤i< 2δ etc.,

$$\begin{aligned} \pi = &\sum_{i=0}^{\delta} \frac{\epsilon(k^{\delta}-1)(k-1)(1-y(\delta-1))}{(k^{\delta}-1)(k^{i+1}-1)} + \\ &\sum_{i=\delta}^{2\delta} \frac{\epsilon(k^{2\delta}-1)(k-1)(1-y(\delta-1))}{(k^{\delta}-1)(k^{i+1}-1)} + \cdots + \\ &\sum_{i=z\delta}^{n} \frac{\epsilon(k^{(z+1)\delta}-1)(k-1)(1-y(\delta-1))}{(k^{\delta}-1)(k^{i+1}-1)} + \epsilon(\delta - 1)y \end{aligned}$$

where z=$\lfloor \frac{n}{\delta} \rfloor$. This is the time taken to generate a reachability graph of height n when the proposed algorithm is used.

3.4.2 The Tree Model

The *Tree Model* is proposed as an enhancement of the Sequential model. From Definition 2, a difference state in latter is expressed as the variation from its immediately preceding state. Although this reduced the memory requirement, regenerating a difference state required backtracking all the way to a state stored in explicit form. Consequently this involves significant costs for regenerating states that were far from an explicit state. To prevent such scenarios, we alter the definition of a difference state.

Definition 4 *Let S_{st} be a state with state S_{ex} as its nearest state in explicit form. The difference form of S_{st}, denoted as D^t_{st}, is defined as the changes necessary in S_{ex} to generate S_{st}.*

Corollary 4 *Let D^t_{st} be the difference form of a state S_{st} with state S_{ex} as its nearest state in explicit form. The state S_{st} can be regenerated in explicit form as S_{st}=S_{ex}+D_{st}.*

Instead of defining a difference state as the changes required in immediately previous state, it is now defined as the changes required in nearest explicit state. This ensures that a state can always be regenerated by backtracking once when the required changes recorded in D^t are applied into the nearest explicit state. However, the nearest explicit state could also be a successor state. Considering that the successor states are not known in advance, the states from ($\lceil \frac{\delta-1}{2} \rceil + 1$) to ($\delta - 1$) are initially stored in explicit form and later transformed when the successor is known.

The difference between Sequential and Tree models is illustrated using Figure 3.13 and 3.14. Figure 3.13 presents a portion of reachability graph for the CPN model in Figure 3.3 using tree model. Initially the model is in State 1 and since it does not have a previous state, it is stored in explicit form. As with Sequential model, the tokens in the model are arbitrarily named *a* to *e* and their place and values are assigned using '→' and '='. However, as compared to Sequential model, States 2, 3 and 4 are stored as their difference from State 1 (supposing *State 1* is the nearest explicit state for *State 2*, *State 3* and *State 4*). Furthermore, contrary to Sequential model, the difference is not always calculated using a previous state. As shown in Figure 3.14, States 4 and 2 might also be stored as the difference from a child state (State 5 in Figure 3.4) if it is deemed to be the nearest explicit state.

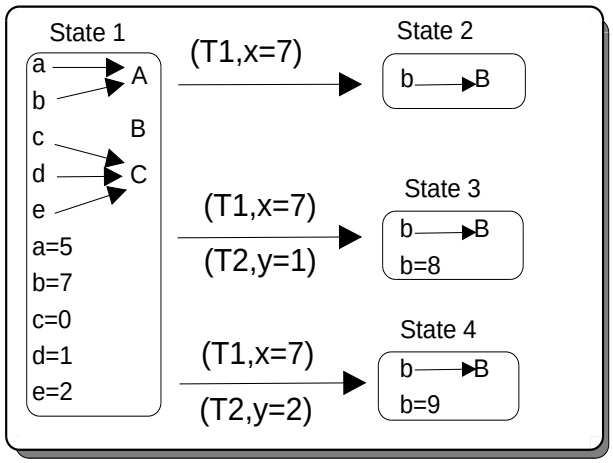


Figure 3.13: A part of reachability graph in Figure 3.4 using tree algorithm

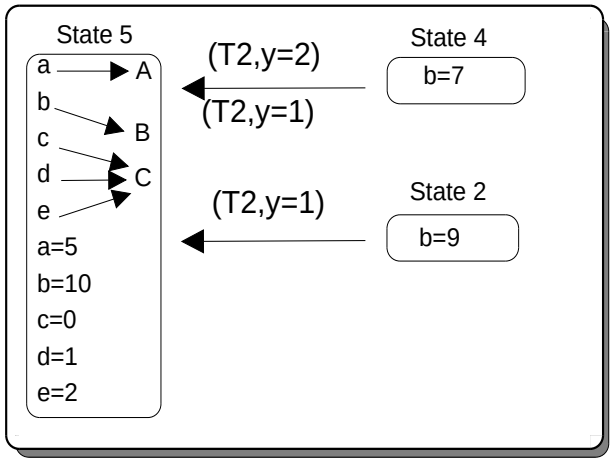


Figure 3.14: The difference form is determined using nearest explicit state, even if latter is a child (or descendent) of the former

On comparing Figure 3.13 with Figure 3.9, certain difference states for Tree model are found to contain more information than their counterparts in Sequence model. This is essentially because the difference between two consecutive states (stored in sequential model) is often less than the states separated by intermediate markings. Consequently the memory reduction offered by Tree model is sometimes less when compared to the Sequential model. However, as emphasized previously, Tree model allows single-step reconstruction of a difference state. Consequently Tree model is more time-efficient when compared to Sequential model and the choice depend on priorities when exploring the state-space.

Figure 3.14 shows that the nearest explicit state need not always be a preceding state. Considering that there are $(\delta-1)$ difference states between any two explicit states S_{before} and S_{after}, the first $\lceil\frac{\delta-1}{2}\rceil$ states have former as their nearest explicit state while the remaining have latter. However, considering that the successor states are not known in advance, the states from $(\lceil\frac{\delta-1}{2}\rceil+1)$ to $(\delta-1)$ are initially stored in explicit form and later transformed when S_{after} is known.

The Tree model uses the nearest explicit states in order to ensure a minimal size for difference states. This is based on the assertion that systems usually change in many small steps, wherein each event inches it away from its previous state towards a terminating state. Consequently a small δ reduces the memory requirement of Tree model. In case this assertion is invalid, either of S_{before} and S_{after} can be used to transform all the $(\delta-1)$ states.

Algorithm for Tree Model

In this section, the algorithm for Tree model is introduced and explained. As with Sequential model, the algorithm for Tree model reduces memory requirement by storing states in difference form. However, it differs in the mechanism used to determine the difference form of a state.

The state-space exploration begins at the initial state of a system and traverses all states that are reachable from it. Each unique state encountered is stored into a hash-table after determining the index using a hash-function. Furthermore, starting with the initial state, each state at depth δ is stored in explicit form. The form in which the remaining states are stored depend on their proximity to explicit states:

- the difference form for states at depth (iδ+1) to (iδ+$\lceil\frac{\delta-1}{2}\rceil$) can be immediately determined using the explicit state at iδ. Consequently they are stored in difference form.
- the difference form for remaining states at depth (iδ+$\lceil\frac{\delta-1}{2}\rceil$+1) to ((i+1)$\delta$-1) cannot be immediately determined as the explicit state at (i+1)δ is unknown. Consequently they are stored in explicit form until (i+1)δ is determined. In order to save time, a buffer of size $\lceil\frac{\delta-1}{2}\rceil$ is used to store these explicit states temporarily.

As compared to Sequential model, the algorithm for Tree model has four parts:

1. `SEARCH:` The steps are listed in Algorithm 4. It accepts a state (S_{st}), it's previous state (S_{pv}) and the distance of S_{st} from last expanded state (depth) as input. A hash function

H is used to find the index for S_{st} as shown in step 1. The algorithm then checks the content of hash-table at this index. There can be again three possibilities.

(a) *Hash table contain NULL at this index:* In this case, it is the first time this state is generated. Consequently algorithm 5 is invoked to store the state at this index. Thereafter any enabled event at this state is fired. Steps 2-4 in algorithm 4 check and handle this case.

(b) *Hash table contain a linked-list at this index:* In this case, each state in the linked-list has hashed to this index. Consequently S_{st} is compared with each state in this list. For explicit states in the list, this comparison is direct. However, difference states need to be expanded before comparison using algorithm 6. If case of a match, S_{st} is neither stored nor analysed for any enabled events. Otherwise algorithm 5 stores the marking at the head of linked list and any enabled event is executed. Steps 5-11 in algorithm 4 check and handle this case.

(c) *Hash table contain a hash-function at this index:* If this case, all states that hashed to this index are stored in a separate hash-table HASHi, indexed by the function H' stored at this index. The index in the second hash-table is calculated in step 14. Thereafter in step 15, this index is checked to determine if it is empty or stores a state. In case it is empty, the state is inserted using algorithm 5 and any enabled event is executed. Steps 15-24 in algorithm 4 handle this case.

2. `INSERT:` The second algorithm is responsible for inserting a state into a hash-table and its steps are listed in algorithm 5. It accepts a state (S_{st}), it's previous state (S_{pv}) and the distance of S_{st} from last expanded state (depth) as input. The first step determines the index of state S_{st} using hash-function. Thereafter the offset from last expanded state (stored in *depth*) is checked. If the depth is 0, the state is stored in expanded form. Furthermore, this instigates calculating the difference form for each of last $\frac{\delta}{2}$ states. As discussed earlier, these states are initially stored in explicit form until their nearest explicit state is determined. Otherwise the difference form for S_{st} is determined. In any cases, the state is stored in pointer "new" that has a field "type" to identify if it is stored in difference or explicit form. It also has a field to store the state itself (either explicit or difference) and a couple of pointers to record the nearest explicit state & the immediately previous state.

   Thereafter the content of hash-table at the index calculated in step 1 is checked. There

Algorithm 4: SEARCH (State S_{st}, int depth,State S_{pv})

```
   Data: State S_st, int depth, State S_pv
   Result: Decide if a marking generated is new
 1 i←H[S_st] ;
 2 if HASH[i]=NULL then
 3 |  INSERT(S_st,depth,S_pv);
 4 |  foreach S_nx such that S_st[(t,c)⟩S_nx do SEARCH(S_nx,(depth+1)mod δ,S_st);
 5 else if HASH[i] points to a linked list then
 6 |  foreach state d in linked list do
 7 |  |  if d is compact then d←RECONSTRUCT(d);
 8 |  |  if d=S_st then return;
 9 |  end
10 |  INSERT(S_st,depth,S_pv);
11 |  foreach S_nx such that S_st[(t,c)⟩S_nx do SEARCH(S_nx,(depth+1)mod δ,S_st);
12 else if HASH[i] contains a hash function then
13 |  H′ ← HASH[i];
14 |  j←H′[S_st] ;
15 |  if HASHi [j] is empty then
16 |  |  INSERT(S_st,depth,S_pv);
17 |  |  foreach S_nx such that S_st[(t,c)⟩S_nx do SEARCH(S_nx,(depth+1)mod δ,S_st);
18 |  else
19 |  |  if HASHi[j] is in compact form then
20 |  |  |  HASHi[j]←RECONSTRUCT(HASHi[j]);
21 |  |  end
22 |  |  if HASHi[j]=S_st then return;
23 |  |  INSERT(S_st,depth,S_pv);
24 |  |  foreach S_nx such that S_st[(t,c)⟩S_nx do SEARCH(S_nx,(depth+1)mod δ,S_st);
25 |  end
26 end
```

could be three possibilities.

(a) *Hash table contain NULL at this index:* In this case, it is the first time this marking is generated. The contents of *new* is simply copied to this index of hash-table. It is worth mentioning that this element is hereafter treated as a linked-list with single node. This allows generalising the handler for hash-collisions. The logic for this case is handled in step 20-21 of algorithm 5.

(b) *Hash table contain a linked-list at this index:* In this case, the contents of *new* is copied to head of linked list to handle hash-collision. Thereafter the list is checked to ensure that it contains not more than 10% of all possible states. Otherwise the

states in it are moved into another hash table and the hash function used is stored at this index. The logic for this case is shown in steps 22-33 of algorithm 5.

(c) *Hash table contain a hash-function at this index:* In this case, the hash-function stored at this index is used to find the position of state in the secondary hash-table and *new* is copied into it. Note that having discrete hash-functions for each index of primary hash-table allows minimizing collisions in the secondary hash table. Steps 35-37 in algorithm 5 handle this case.

3. `RECONSTRUCT`: The third algorithm accepts a state in difference form and returns its explicit form. The transformation basically involves applying the differences into the nearest explicit state. The steps are listed in algorithm 6.

Comparison with Sequential Model

The proposed algorithm for Tree model promises to reduce the amount of space necessary to store the states by using difference states. Furthermore, it eliminates any associated delay due to backtracking. In this section, we evaluate the Tree model by comparing its time and memory requirements with the Sequential model.

Comparing Percentage increase in Memory for Tree model: Suppose the memory occupied by an explicit state is λ, while the average difference between consecutive states is $x*\lambda$, where $0<x<1$. In case of sequential model, each explicit state would occupy a space λ and each difference state would occupy $x*\lambda$. While the former remains same for Tree model, the latter changes because the difference states contain the difference from the nearest explicit state instead of immediately previous state. In the worst case, the difference of a state from nearest explicit state would be sum of differences stored in intermediate states. Therefore the memory needed to store the difference states between any two explicit states would be

$$2*(x*\lambda+2*x*\lambda+3*x*\lambda+\cdots+\tfrac{\delta}{2}*x*\lambda) \text{ if } \delta \text{ is even}$$
$$2*(x*\lambda+2*x*\lambda+3*x*\lambda+\cdots+\tfrac{\delta-1}{2}*x*\lambda)+\ \tfrac{\delta+1}{2}*x*\lambda \text{ if } \delta \text{ is odd}$$

These equations can be simplified into

$$x*\lambda * \tfrac{\delta(\delta+2)}{4} \text{ if } \delta \text{ is even}$$
$$\text{and } x*\lambda * \tfrac{(\delta+1)^2}{4} \text{ if } \delta \text{ is odd}$$

Algorithm 5: INSERT(State S_{st}, int depth, State S_{pv})

```
   Data: State S_st,int depth, State S_pv
   Result: Insert State S_st into hash table
 1 i← H[S_st] ;                                  // H is used to calculate the index
 2 if depth=0 then                          // when depth is 0, store in explicit form
 3 |   new.type←explicit;
 4 |   new.state←S_st;
 5 |   for i=1; i≤ δ/2;i++,S_pv=S_pv.prev do
 6 |   |   S_pv.state←S_st-S_pv.state;     // difference form for δ/2 previous states
 7 |   |   S_pv.near←S_st;          // that were temporarily stored in explicit form
 8 |   end
 9 |   nearest←S_st;
10 else if depth≥ δ/2 then
11 |   new.type←explicit;                    // store temporarily is explicit form
12 |   new.state←S_st;
13 |   new.prev←S_pv;
14 else
15 |   new.type←difference;                          // store in difference form
16 |   new.state← S_st-nearest;
17 |   new.near←nearest;
18 |   new.prev←S_pv;
19 end
20 if HASH[i]=NULL then
21 |   HASH[i]=new;                                                  // No Conflict
22 else if HASH[i] points to a linked list then
23 |   insert new at the head of linked list;  // Multiple states hash to same index
24 |   if length(linked list)≥ |M|/10 then  // if length is longer than 10 percent
25 |   |   foreach state d in linked list do              // create a new hash table
26 |   |   |   if d is in compact form then
27 |   |   |   |   d1←d;
28 |   |   |   |   d←RECONSTRUCT(d);                    // get explicit state
29 |   |   |   end
30 |   |   |   j←H[d] ;
31 |   |   |   add d1 to HASHi[j];                // store in secondary hash table
32 |   |   end
33 |   end
34 else if HASH[i] points to a hash function then
35 |   H′ ← HASH[i];                              // get secondary hash function
36 |   j←H′[S_st] ;
37 |   HASHi[j]←new;
38 end
```

Algorithm 6: RECONSTRUCT(State d)

Data: State d in difference form
Result: Explicit form of d is returned
1 return d.state+d.near;

As with Sequential model, we assume that the average number of new states generated by a transition is k (k>1). The number of states at depth d is given by k^d as illustrated in Figure 3.12. All dark circled represent explicit states while shaded circles represent difference states. The number of expanded states in a reachability of height n is the sum of the number of expanded states at depth 0, δ, 2δ, $\cdots$, $\lfloor \frac{n}{\delta} \rfloor * \delta$. This is given by

$$\beta_{expanded} = k^0 + k^{\delta} + k^{2\delta} + \cdots + k^{\lfloor \frac{n}{\delta} \rfloor * \delta}$$

This is a geometric progression [Bronshtein et al., 1997] with initial term a=1 and ratio r=k^{δ}. Hence, the sum is given by

$$\beta_{expanded} = \frac{a(r^{n+1}-1)}{r-1} = \frac{k^{(\lfloor \frac{n}{\delta} \rfloor + 1)*\delta}-1}{k^{\delta}-1}$$

The number of explicit states for Tree model is same as Sequential model and therefore there is no extra memory needed to store them. From the previous discussion, the difference states at depth i$*\delta$-1 and i$*\delta$+1 require space x$*\lambda$, which is again same as for Sequential model. However, the difference states at other depths account for the additional memory requirement.

Between any two explicit states at depth i$*\delta$ and (i+1)$*\delta$, the difference in memory requirements for Tree and Sequential models are

$$\begin{gathered}
\partial_{i*\delta+2} = 2*x*\lambda*k^{i*\delta+2} - x*\lambda*k^{i*\delta+2} = x*\lambda*k^{i*\delta+2} \\
\partial_{i*\delta+3} = 3*x*\lambda*k^{i*\delta+3} - x*\lambda*k^{i*\delta+3} = 2x*\lambda*k^{i*\delta+3} \\
\vdots \\
\partial_{i*\delta+\frac{\delta}{2}} = \frac{\delta}{2}*x*\lambda*k^{i*\delta+\frac{\delta}{2}} - x*\lambda*k^{i*\delta+\frac{\delta}{2}} = (\frac{\delta}{2}-1)*x*\lambda*k^{i*\delta+\frac{\delta}{2}} \text{ if } \delta \text{ is even}
\end{gathered}$$

We take advantage of the symmetry of difference around $\frac{\delta}{2}$ and evaluate the difference only for the first half of difference states. The remaining states are accounted by doubling sum of above differences, which is evaluated as

$$\begin{gathered}
\partial = x*\lambda*k^{i*\delta+2} + 2x*\lambda*k^{i*\delta+3} + \cdots + (\frac{\delta}{2}-1)*x*\lambda*k^{i*\delta+\frac{\delta}{2}} \text{ if } \delta \text{ is even} \\
\text{or } \partial = x*\lambda*k^{i*\delta+1}*(k+2k^2+\cdots+(\frac{\delta}{2}-1)k^{\frac{\delta}{2}-1}) \quad (7)
\end{gathered}$$

In order to simplify the equation, we assign the part in brackets to E.

$$\text{E=k+2k}^2\text{+}\cdots\text{+}(\tfrac{\delta}{2}\text{-1})\text{k}^{\frac{\delta}{2}-1}$$
$$\text{or k}*\text{E=k}^2\text{+2k}^3\text{+}\cdots\text{+}(\tfrac{\delta}{2}\text{-2})\text{k}^{\frac{\delta}{2}-1}\text{+}(\tfrac{\delta}{2}\text{-1})\text{k}^{\frac{\delta}{2}}$$
$$\therefore \text{E-k}*\text{E=k+k}^2\text{+k}^3\text{+}\cdots\text{+k}^{\frac{\delta}{2}-1}\text{-}(\tfrac{\delta}{2}\text{-1})\text{k}^{\frac{\delta}{2}}$$
$$\text{or E(1-k)=}\tfrac{k*(k^{\frac{\delta}{2}}-1)}{k-1}\text{-}(\tfrac{\delta}{2}\text{-1})\text{k}^{\frac{\delta}{2}}$$
$$\text{or E=}\tfrac{1}{k-1}((\tfrac{\delta}{2}\text{-1})\text{k}^{\frac{\delta}{2}}\text{-}\tfrac{k*(k^{\frac{\delta}{2}}-1)}{k-1})$$

Assigning E in equation 7 we get

$$\partial\text{=x}*\lambda*\text{k}^{i*\delta+1} * \tfrac{1}{k-1}((\tfrac{\delta}{2}\text{-1})\text{k}^{\frac{\delta}{2}}\text{-}\tfrac{k*(k^{\frac{\delta}{2}}-1)}{k-1})$$

The value in ∂ gives the additional memory needed to store the difference states between i$*\delta$+2 to i$*\delta$+$\frac{\delta}{2}$. Due to symmetry around $\frac{\delta}{2}$, the additional memory needed to store difference states between i$*\delta$+2 and (i+1)$*\delta$-2 is 2$*\partial$. The memory needed by Sequential model to store these difference states is

$$\text{x}*\lambda*(\delta\text{-3})$$

Therefore percentage increase in memory requirement for Tree model is given as

$$\tfrac{2*\partial}{x*\lambda*(\delta-3)}\%$$
$$\text{or } \tfrac{2*x*\lambda*k^{i*\delta+1}*\frac{1}{k-1}((\frac{\delta}{2}-1)k^{\frac{\delta}{2}}-\frac{k*(k^{\frac{\delta}{2}}-1)}{k-1})}{x*\lambda*(\delta-3)}\%$$
$$\text{or } \tfrac{2*((\frac{\delta}{2}-1)k^{\frac{\delta}{2}}-\frac{k*(k^{\frac{\delta}{2}}-1)}{k-1})}{2*(\delta-3)*(k-1)}\ \%$$

Comparing Percentage Reduction in Time: As discussed earlier, Tree model eliminates any delay due to backtracking. Consequently this comparison only considers the backtracking delays as any other delay is common for both Sequential and Tree models.

Let i be an integer between 1 and n. If the height of reachability graph is i, the number of states in explicit and difference forms are given by

$$\beta'_{total} = \tfrac{k^{i+1}-1}{k-1},\ \beta'_{expanded} = \tfrac{k^{(\lfloor\frac{i}{\delta}\rfloor+1)*\delta}-1}{k^{\delta}-1}$$
$$\text{and } \beta'_{difference} = \tfrac{k^{i+1}-1}{k-1} \text{ - } \tfrac{k^{(\lfloor\frac{i}{\delta}\rfloor+1)*\delta}-1}{k^{\delta}-1}$$

When a state S_{st} is generated, it is compared with the state stored at an index given by the hash-function. The probability that this state is stored in expanded or difference form can be calculated as

$$\mathrm{P}_{expanded} = \frac{\beta'_{expanded}}{\beta'_{total}} \text{ and } \mathrm{P}_{difference} = \frac{\beta'_{difference}}{\beta'_{total}} \quad (8)$$

If the state is stored in difference form, it has to be first expanded by backtracking and then compared with S_{st}. Hence, time taken for comparing S_{st} with the stored states is given by

$$\mathrm{T}_{comparison} = \mathrm{T}_{expanded} + \mathrm{T}_{difference}$$

Suppose the time for comparing two expanded states is ϵ, while it takes y$*\epsilon$ time for backtracking a single step. While Tree model always requires backtracking a single step, Sequential model would need a backtracking of $(\delta - 1)$ steps in worst case for expanding the state. Therefore the time taken for the two models be calculated as

$$\mathrm{T}^{seq}_{comparison} = \mathrm{P}_{expanded} * \epsilon + \mathrm{P}_{difference} * \epsilon(\delta - 1)y$$
$$\& \ \mathrm{T}^{tree}_{comparison} = \mathrm{P}_{expanded} * \epsilon + \mathrm{P}_{difference} * \epsilon * y$$

Consequently the difference in time taken for each comparison would be

$$\mathrm{T}^{seq}_{comparison} \text{ - } \mathrm{T}^{tree}_{comparison}$$
$$=(\mathrm{P}_{expanded} * \epsilon + \mathrm{P}_{difference} * \epsilon(\delta - 1)y)\text{-}(\mathrm{P}_{expanded} * \epsilon + \mathrm{P}_{difference} * \epsilon * y)$$
$$=\mathrm{P}_{difference} * \epsilon(\delta - 2)y$$
$$= \frac{\beta'_{difference}}{\beta'_{total}} * \epsilon(\delta - 2)y \text{ from equation 8}$$
$$= (1\text{-}\frac{\beta'_{expanded}}{\beta'_{total}})*\epsilon(\delta - 2)y$$
$$=\epsilon(\delta - 2)y\text{-}\frac{\beta'_{expanded}}{\beta'_{total}} * \epsilon(\delta - 2)y$$
$$=\epsilon(\delta - 2)y\text{-}\frac{(k^{(\lfloor \frac{i}{\delta} \rfloor + 1)*\delta} - 1)(k-1)}{(k^{i+1}-1)(k^{\delta}-1)} * \epsilon(\delta - 2)y$$

This is the additional time taken by Sequential model for comparing a state generated with a stored state in difference form. All comparison at a particular depth takes place concurrently. Hence the additional time taken to generate reachability graph of height n is the sum of extra time taken for one comparison at each level. This is denoted by ∂', where

$$\partial' = \epsilon(\delta - 2)y\text{-}\epsilon(\delta - 2)y \sum_{i=0}^{n} \frac{(k^{(\lfloor \frac{i}{\delta} \rfloor + 1)*\delta} - 1)(k-1)}{(k^{\delta}-1)(k^{i+1}-1)}$$

Since $\lfloor \frac{n}{\delta} \rfloor$=0 for 0≤i< δ, $\lfloor \frac{n}{\delta} \rfloor$=1 for δ ≤i< 2δ etc.,

$$\partial' = \epsilon(\delta - 2)y\text{-}\epsilon(\delta - 2)y \sum_{i=0}^{\delta} \frac{(k^{\delta}-1)(k-1)}{(k^{\delta}-1)(k^{i+1}-1)} +$$
$$\epsilon(\delta - 2)y \sum_{i=\delta}^{2\delta} \frac{(k^{2\delta}-1)(k-1)}{(k^{\delta}-1)(k^{i+1}-1)} + \cdots +$$
$$\epsilon(\delta - 2)y \sum_{i=z\delta}^{n} \frac{(k^{(z+1)\delta}-1)(k-1)}{(k^{\delta}-1)(k^{i+1}-1)} +$$

where z=$\lfloor \frac{n}{\delta} \rfloor$. This is the additional time taken by Sequential model to generate a reachability graph of height n.

This section theoretically evaluated the Tree model by comparing it with the Sequential model. In the next section, the Sequential method is experimentally evaluated.

3.5 Experimental Results

The proposed algorithm for Sequential model was tested on a desktop with 2.8GHz Intel Pentium D processor and 1GB RAM. The desktop had Ubuntu 8.04 desktop version OS installed and our C source code was compiled using GNU C compiler(gcc).

We used six different Coloured Petri-net models to run our experiment. The number of places and tokens in each CPN model is listed in Table 3.3 and Table 3.4. If a model had m tokens and n places, each token was assigned an integer name i:i∈[0,m-1] and each place was assigned an integer name j:j∈[0,n-1]. Initially, all tokens were in place 0. At each state, the set of enabled transitions were selected randomly and one of these transitions was fired. This allow having a large number of transitions in a model without specifying the bindings for which they are enabled.

For each CPN model, we have calculated the time and space needed to generate first 500 unique states using Sequential model and without using it. Furthermore, Sequential model require a non-negative integer value of δ and we have assigned it the set of values {1,2,3,7,20}. When the sequential algorithm is not used, we assign 0 to δ. The results are listed in Table 3.3 and Table 3.4. δ is the shortest distance between two explicit states.

Table 3.3 shows the memory needed (given by Λ) to store first 500 states of CPN models used and the percentage reduction in memory requirement (given by Δ) for different values of δ. For each model, the memory requirement is highest either when not using sequential algorithm (δ=0), or when using it with δ=1. In Figure 3.15, memory required is plotted against value of δ. On increasing the value of δ, the memory requirement decrease for all models. Furthermore, the decrease is significantly higher for large models, with a significantly greater number of places and tokens, as compared to small models. For instance, the CPN model with 1500 places and 2000 tokens used 95% less space when sequential algorithm was used with δ=20. Compared to this, the reduction was 76% for a CPN model with 4 places and 5 tokens. Nevertheless, the reduction is massive for models of all sizes and for all values of δ, as evident from Table 3.3 and Figure 3.15.

Table 3.4 shows the time needed (given by π) to generate the first 500 states of CPN

Table 3.3: Space occupied (in bytes) by first 500 states of CPN models (Λ) and percentage decrease in space (Δ). n and m are the number of places and tokens in these models.

n	m	δ=0	δ=1		δ=2		δ=3	
		Λ	Λ	Δ	Λ	Δ	Λ	Δ
4	5	9980	9980	0%	5996	40%	4668	53%
60	90	179640	179640	0%	90996	49%	61448	66%
200	400	798400	798400	0%	400996	50%	268528	66%
400	700	1397200	1397200	0%	700996	50%	468928	66%
800	1000	1996000	1996000	0%	1000996	50%	669328	66%
1500	2000	3992000	3992000	0%	2000996	50%	1337328	67%

n	m	δ=7		δ=20	
		Λ	Δ	Λ	Δ
4	5	3148	68%	2396	76%
60	90	27628	85%	10896	94%
200	400	116908	85%	41896	95%
400	700	203308	85%	71896	95%
800	1000	289708	85%	101896	95%
1500	2000	577708	86%	201896	95%

models used and the percentage increase in delay (given by η) for different values of δ. For each model, the delay is minimum when sequential algorithm is not used (δ=0). In Figure 3.16, delay is plotted against value of δ. When sequential algorithm is used, delay increase with increase in value of δ. A model with 4 states and 5 tokens generates 500 states almost instantly (π=0) without using the sequential algorithm. The same model needs 1.5 second when sequential algorithm is used with δ=20. The percentage increase in delay (η) decrease with an increase in size of model. The model with 1500 places and 2000 tokens has twice the delay when sequential algorithm is used with δ=20 as compared to when δ=0 (algorithm not used). A comparison of results in Table 3.3 and Table 3.4 clearly shows that the reduction in memory requirement comes at the cost of extra delay in processing. This is further evident in Figure 3.17 where the required memory decrease and delay increase with increase in value of δ. However, the envisioned memory reduction is massive as compared to the increase in delay, especially for moderate to large size models. A model with 1500 places and 2000 tokens had 95% reduction in memory with double the delay. Due to inherent complexity of most modern systems, their models are almost always large. The sequential and tree models are addressing a niche for such systems.

Table 3.4: Time (in msec) to generate first 500 states of CPN models (π) and percentage increase in time (η). n and m are the number of places and tokens in these models.

n	m	δ=0	δ=1		δ=2		δ=3	
		π	π	η	π	η	π	η
4	5	$\approx$0	20	∞	90	∞	160	∞
60	90	20	40	100%	80	300%	100	400%
200	400	80	130	62%	170	112%	200	150%
400	700	140	220	57%	260	85%	290	107%
800	1000	200	310	55%	350	75%	380	90%
1500	2000	400	620	55%	660	65%	680	70%

n	m	δ=7		δ=20	
		π	η	π	η
4	5	430	∞	1490	∞
60	90	210	950%	570	2750%
200	400	320	300%	680	750%
400	700	410	193%	770	450%
800	1000	500	150%	850	325%
1500	2000	820	105%	1200	200%

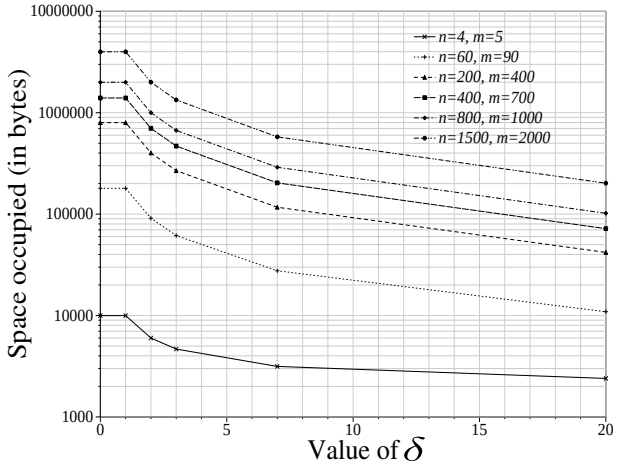


Figure 3.15: Memory requirement decrease with increase in value of δ, the distance between two explicit markings. δ=0 means algorithm not used. Y-axis uses log scale.

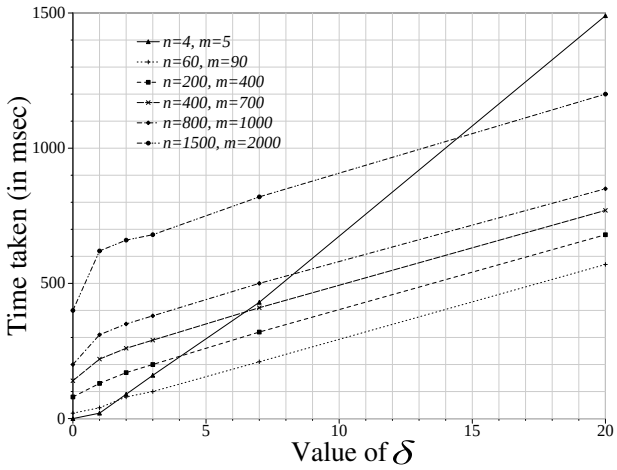


Figure 3.16: Delay increase with increase in value of δ. δ=0 when algorithm not used.

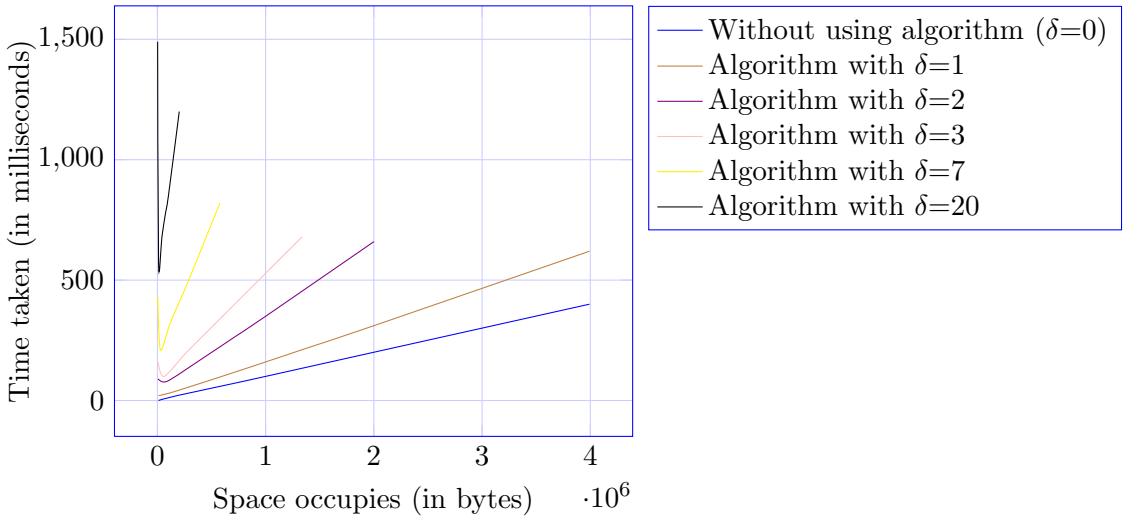


Figure 3.17: Delay increase and memory requirement decrease with increase in value of δ. Each curve is for a different value of δ as indicated by the legend. The six points on a curve correspond to six CPN models used.

3.6 Discussion

The sequential model reduces the memory requirements for model-checking by storing states in difference form and thereby allow model-checking in a machine with a fraction of memory needed otherwise. This might lead to wider use of model-checking in software verification and subsequent production of reliable software systems. Although there is an increase in delay due to backtracking, the results illustrate that the delay is small when compared to the massive reduction in memory obtained.

Reduction in memory requirement Λ: The Sequential model is found to increasingly reduce the memory requirement with an increase in value of δ. State-space analysis without using sequential algorithm will store all states in explicit form, leading to maximum memory requirement. This holds for our results in Figure 3.15. Furthermore, using sequential algorithm with δ=1 also stores all states in explicit form, keeping Λ unchanged. However when δ=2, every alternate state is stored in explicit form. This leads to almost 50% reduction in Λ as only half the total number of states are in explicit form. Similarly, when δ=3, one in every three states are stored in explicit form leading to 66% reduction in Λ. When δ=7, one in seven states is stored in explicit form leading to 85% reduction in Λ. Finally, when δ=20, one in 20 states is stored in explicit form resulting in 95% reduction in value of Λ.

The reduction for CPN model with 4 places and 5 tokens is low as compared to other models. The reason being that the size of an explicit state is almost same as a difference state for a small model. Therefore, replacing explicit state with difference state do not make a big difference.

Increase in delay π: The Sequential model is found to increase the delay with an increase in value of δ and a corresponding decrease in memory requirement. Two factors contribute to overall delay: 1) backtracking to expand a difference state 2) when state-space is being explored using DFS algorithm and a duplicate state is encountered, the stack is popped till a state with an enabled event (transition) is encountered. Popping a stack is time intensive operation.

A small model has less number of possible states and therefore the chances of encountering a duplicate state is high. The CPN model with 4 places and 5 tokens encountered 469 duplicate states before generating 500^{th} unique state. As compared to this, the model with 60 places and 90 tokens encountered only 1 duplicate state before generating 500^{th} state. The

delay in popping stack, combined with backtracking delay lead to large π for small models.

When model-checking, we need not backtrack if all states are in explicit form. This leads to low π when there are no difference state (δ=0 or δ=1). However, due to extra processing delay of sequential algorithm, the delay for δ=1 is higher than δ=0. On further increasing δ, delay increases due to backtracking. The higher the value of δ, more is the backtracking needed to expand a state and greater the delay.

Reducing the cost of model-checking would encourage its wider use in the software development life-cycle. This in turn would enhance the reliability and safety of software systems. Considering our widespread dependency on such systems (e.g. traffic signals, elevators) this would also ensure our safety and well-being.

3.7 Summary

This chapter reduced the memory requirement for model-checking by storing states in difference form. Consequently, model-checking would acquire a bigger role in verification of a wide range of software. This ensures safety and reliability of software systems and enhances their correctness. Experimental results indicate that our models require significantly less memory to verify a software system. Furthermore the proposed technique is found to perform better with larger models. Contemporary systems have high level of complexity, often leading to large models. The proposed models are addressing a niche for such systems.

Chapter 4

Time Efficient State-Space Analysis in Software Model-Checking

The previous chapter identified the significance of formal methods in verifying service compositions and the massive time and memory costs associated with it. Considering the ubiquity of software systems in our daily life, the previous chapter vouched for wider use of formal methods to warrant their correctness and reliability. *Sequential* and *Tree* models were proposed to pursue this inducement by reducing the memory costs involved in *model-checking*, a widely used formal method. However, as with any memory reduction technique, the models were found to have an associated time overhead.

In this chapter, we seek to reduce the aforementioned delay by introducing concurrency into the paradigm of model-checking. Contemporary model-checking languages offer different levels of abstraction by defining a notion of hierarchy, wherein a system is modeled as a set of interdependent modules. The offered reduction in time is attributed to the concurrent exploration of all such modules in a hierarchical model and exposing the outcome using special data-structures. This allows the modules to interact with each other and resolve their dependencies when generating the state-space. Our experiments report a time reduction of 86% in generating the first 25,000 markings. Furthermore, the offered reduction increases as more markings are generated. As compared to the recent solutions that depend on the existence of *stubborn sets* and/or *symmetry* in the state-space, our technique only necessitates a hierarchical model. Considering that most modern modeling languages have incorporated the notion of hierarchy, this is a fairly lenient prerequisite.

4.1 Motivation

The previous chapter identified the significance of formal methods in verifying service compositions and the massive time and memory costs associated with it. Thereupon we proposed *Sequential* and *Tree* models to reduce the memory costs for model-checking. In this chapter we propose a novel method for reducing the time requirements for model-checking so that they could be more widely used.

The delay in model-checking could be attributed to a range of factors. A significant delay could be accredited to the primary job of MC wherein each state of a system is checked for a set of undesirable properties. This delay is formidable for systems that exhibit state-space explosion problem. This is further exacerbated by the delay in storing states of a system explored hitherto by the model-checker and comparing them with the states to be produced hereafter. As observed in the previous chapter, certain states of the system might be generated repeatedly and prevent the model-checker from terminating. Consequently it is necessary to store and compare states in order to identify duplicate states and ensure the termination of model-checking process. Regardless of the data-structure used for storage and the efficiency of comparison, there is an associated time overhead. This is in addition to the delay considered earlier in generating and processing a state. As observed earlier, the state-space explosion problem could further aggravate the delay. Furthermore, considering the cost and stringent upper bounds (based on architecture, OS, processor, motherboard etc.) of memory in a system, there has been extensive research in reducing the memory requirements for storing states. Unfortunately, as seen in previous chapter, all attempts in reducing the memory requirements for model-checking are accompanied with a 'doubling or tripling' of time requirements [Evangelista and Pradat-Peyre, 2005]. This provides further inducement in proposing a technique for reducing the delay that could be used either independently or with a memory-reduction algorithm.

Some solutions based on 'Partial Order Reduction' [Godefroid, 1996] address this problem by exploiting the independence of concurrently executing events. However, detecting such independent events might sometimes be as difficult as the underlying verification problem [Gueta et al., 2007]. A few other solutions are based on detecting 'Symmetry' in the state space [Emerson and Sistla, 1996] wherein the state-space is partitioned into equivalent classes corresponding to isomorphic graphs and one state is used to represent each class. However, this requires determining the representative state (known as the 'orbit problem' [Clarke et al., 1998; Emerson and Sistla, 1996]) which is at least as hard as the 'graph isomorphism problem'

[Clarke et al., 1998; Köbler et al., 1993].

In this chapter, we propose a novel method to reduce the time requirement for model-checking a hierarchical model. A hierarchical-model consists of a set of inter-dependent modules. We explore each of these modules in parallel to generate its *Parametrised Reachability Graph* (PRG) and *Access-tables* that act as a repository of corresponding module behaviour. Thereafter a module can use these data-structures to determine the behaviour of any other module without actually executing it. In addition to concurrency, exposing such a module behaviour repository for each module helps in reducing the delay. For each module, the dependency of other modules on it is injected into its repository using parameters. Later these parameters are assigned specific values to obtain the corresponding *reachability graph* for the hierarchical-model.

As discussed previously, the first step in model-checking involves modeling the target system. This can be accomplished using an array of available modeling languages that offer varying convenience in modeling a system. These languages exhibit different levels of elegance and expressiveness. Among all offerings, the notion of hierarchy in a modeling language sets it apart. Other than stepwise refinement and different levels of granularity, a hierarchical modeling language allows sharing and reusing modules [Alur, 2004]. Some common hierarchical languages along with their supported model-checking tool are SMV [McMillan, 2000] for CMU SMV, v-promela [Leue and Holzmann, 1999] for SPIN, Hierarchical timed-automata [David and Möller, 2001] for UPPAAL and CPN-ML for Coloured Petri-net (CPN) Tools [Jensen et al., 2007]. The subtle difference between each of these languages forbid us from proposing an 'all-inclusive' algorithm; the technique proposed in this chapter specifically targets CPN models. However, we do not claim any advantage in using CPN-ML over other modeling languages. The method can be adapted for other languages that have an underlying notion of hierarchy.

Our contributions can be summarised as:

1. We introduce a technique to concurrently explore the modules in a hierarchical model and produce a repository of their behaviour. Any dependency of other modules on it is epitomized as a set of parameters. Such concurrent exploration helps to reduce the time requirement. Furthermore the repository allows determining a module behaviour without actually executing it.

2. We propose a related technique to assign specific values to the parameters in each PRG and generate the reachability graph for the hierarchical model. Assigning values to the

parameters helps in resolving dependency between modules.

The remainder of the chapter is organised as follows. Section 4.2 introduces the deliberated problem and provides an insight into the tendered solution. Thereafter Section 4.3 presents an overview of Hierarchical Coloured Petri-Net model that is later used in discussing the solution. Prior to proposing the technique for time efficient state-space analysis in Section 4.5, the related works are compiled in Section 4.4. The experimental results are presented in Section 4.6 and the outcome is discussed in Section 4.7. Finally we summarize our contributions in Section 4.8.

4.2 An Overview of the Deliberated Problem & the Tendered Solution

In this section, we discuss the problem in detail and outline the proposed solution. State-space analysis of a model is done by generating a reachability graph wherein each node (or state) of the graph is scrutinized to determine if the set of undesirable and/or desirable properties hold. Depending on the number of properties to be analysed at each state, there is an associated time overhead. Furthermore additional delay is incurred in determining the set of enabled events at each state and taking turns in executing them.

The ever-increasing intricacies in a contemporary software systems snowballs the reachability graph to contain a gigantic number of states. This phenomenon, better known as the *state-space explosion* problem [Christensen et al., 2001], leads to exorbitant delays in producing and analysing the state-space. This is further undermined by the necessity of storing states that were generated hitherto and comparing them to any state produced henceforth (as discussed in Section 3.2 of Chapter 3). Regardless of the numerous ingenious algorithms proposed for an efficient storage and comparison of states, there is always an associated time overhead. Still worse, all attempts for a memory efficient storage are accompanied by an increase in delay [Mukherjee et al., 2010; Evangelista and Pradat-Peyre, 2005]. Unfortunately the sheer volume of research on memory-efficiency suggests that reducing the memory-costs have always been a priority, even when this always has an associated time delay. We recognize the necessity of reducing both the memory and time requirements. In our previous work [Mukherjee et al., 2010], we proposed Sequential and Tree models to address the memory costs in model-checking. This chapter proposes a technique for reducing the time-requirement that could be used either independently or with a memory-reduction algorithm.

As pointed out previously, a system needs to be modeled using one of the several available modeling languages prior to generating its state-space. These languages offer different levels

of abstraction, elegance and expressiveness. Among all offerings, the notion of hierarchy in a modeling language sets it apart. A hierarchical model consists of a set of interdependent modules and the envisioned reduction in delay is attributed to the exploration of these modules in parallel.

Most of the existing hierarchical formalisms are limited to modeling a system (e.g. hierarchical Coloured Petri nets [Jensen, 1996], hierarchical state machines [Alur and Yannakakis, 2001]). Prior to generating the state-space, a hierarchical model is usually flattened for convenience. Despite being a legitimate solution, there is no parallelism and concurrency in the process of generating the state-space. The technique proposed herein preserves the hierarchical structure and the individual modules of the model when generating the state-space. Furthermore, in order to reduce the delay associated with generating the reachability graph, these modules are explored in parallel. As a result of this exploration, an access-table and a parametrised reachability graph is obtained for each module that acts as a repository of module behaviour. The parametrised reachability graph for a module contains a set of parameters that epitomises the dependency on other modules on it. Considering that the behaviour of all modules in a hierarchical model are available in these repositories, the reachability graph can be produced by assigning appropriate values to these parameters and resolving the intermodule dependencies.

4.3 An Overview of Hierarchical Coloured Petri-Nets

A *Hierarchical Coloured Petri-Net* (HCPN) [Jensen, 1996] model consists of a finite set of non-hierarchical CPN models, also known as *modules*. Figures 4.1-4.5 represent a collection of modules that together constitute a HCPN model. The modules depict a simple stop-and-wait *Protocol* [Tanenbaum, 2002] for transferring a number of data packets from *Sender* to *Receiver* over an unreliable *Network* (module names in italics). In case of multiple receivers, as is the case here, a copy of the packet is delivered independently to each receiver. The three receivers herein are identified as Recv(1), Recv(2) and Recv(3). The technique for time-efficient state-space analysis, proposed later, is discussed in reference to this example net.

Definition 5 *A module [Jensen and Kristensen, 2009] $s \in S$ is a four-tuple $s=(CPN^s, T^s_{sub}, P^s_{port}, PT^s)$ where,*

1. *$CPN^s = (P^s, T^s, A^s, \sum^s, V^s, C^s, G^s, E^s, I^s)$ is a non-hierarchical Coloured Petri-Net.*

2. $T^s_{sub} \subseteq T^s$ *is a set of substitution transitions.*

3. $P^s_{port} \subseteq P^s$ *is a set of port-places.*

4. $PT : P^s_{port} \rightarrow \{IN, OUT, I/O\}$ *is a port-type function that specify the port-type for each port-place.*

For all s_1, $s_2 \in S : s_1 \neq s_2$, *we have* $(P^{s_1} \bigcup T^{s_1}) \bigcap (P^{s_2} \bigcup T^{s_2}) = \emptyset$. *This implies that no two modules should share their places and/or transitions.*

Corollary 5 *A module containing a substitution transition is known as supermodule, while the module it substitutes is known as submodule.*

Figure 4.1 illustrates the Protocol module containing 6 places (the ellipses) and 3 substitution transitions (bordered rectangles) connected by arcs (arrows). As the name emphasises, a *substitution transition* acts as a substitute for another module that executes whenever it fires. The module substituted is indicated by a tag associated with it. For instance, the substitution transition *Sender* replaces a module by the same name that is shown in Figure 4.2. Similarly the module *Network* in Figure 4.3 has two substitution transitions, each with an associated tag *Transmit*. However, it is important to point out that they replace different copies (or instances) of module *Transmit*. The real power of HCPN lies in the fact that a module can have multiple instances, one for each substitution transition.

Definition 6 *A substitution transition* $t \in T^s_{sub}$ *is defined as three-tuple* $t = \{SM^s, P^t_{sock}, PS^s\}$ *where,*

1. $SM^s : T^s_{sub} \rightarrow S$*-s is the submodule function for module s and associates a module (other than s) with each substitution transition* $t \in T^s_{sub}$. *If* $s' = SM^s(t)$, *then* $s, s' \in S : s \neq s'$ *and there exists a non-trivial path* $s \xrightarrow{t} s'$. *The module* s' *is denoted as the submodule of s.*

2. $P^t_{sock} \subseteq P^s$ *is the set of adjoining places of t and are referred to as its 'socket'.*

3. $PS^s(t) \subseteq P^t_{sock} \times P^{SM^s(t)}_{port}$ *is a port-socket function that is used to 'glue' a module and its submodule. This is accomplished by mapping each port in submodule with a socket in corresponding module.*

A supermodule and its submodules are glued by defining a one-to-one relationship between a subset of their places. Each place adjacent to a substitution transition is known as *socket* and has a counterpart in the associated submodule that is known as *port*. Although

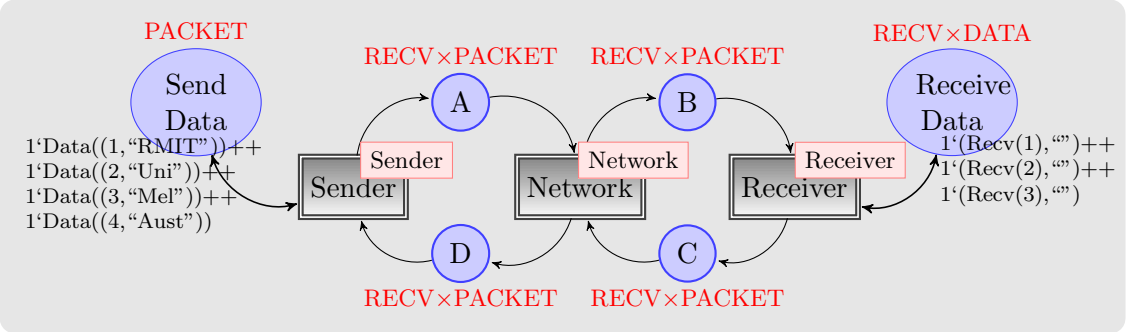


Figure 4.1: Module for the protocol

not necessary, the port-socket pairs in our example net have the same name. Furthermore, considering that a socket could either be an input, output or I/O place of the substitution transition, an equivalent tag is attached to the corresponding port to indicate its permitted type.

If each module is linked to its submodules using directed arcs, the resultant graph obtained is known as *module-hierarchy*. Figure 4.7 illustrates the module-hierarchy of our example HCPN model. Each directed arc in Figure 4.7 is labelled with the name of substitution transition that represents its target module. Considering that a module cannot be its own submodule, the module hierarchy is essentially an acyclic directed graph [Bondy and Murty, 2008]. The roots of the module hierarchy with no incoming arcs are known as *prime modules* [Jensen and Kristensen, 2009] and are denoted by S_{PM}. A module hierarchy can have multiple prime modules.

Each place in a module has an affiliated data-type (known as its colour) and all data-values (known as *tokens*) in it must confirm to this type. For instance the token *1‘Data((1, “RMIT”))* in place *Send Data* is of type *PACKET*. The preceding number followed by a back-quote (e.g. 1‘) indicates the count of a token while a ‘++’ separate two different tokens.

A module might also have non-hierarchical transitions (rectangles with no border) that can manipulate these data values when *enabled* and move them between places through an arc. In order to enable a transition, an appropriate value needs to be assigned to each variable associated with it, and this is known as *binding*. A binding enables a transition iff 1) each arc expression on input-arc evaluate to a subset of marking of the corresponding input place 2) guard condition (if exists) is satisfied. Accordingly a binding b enables a transition t if

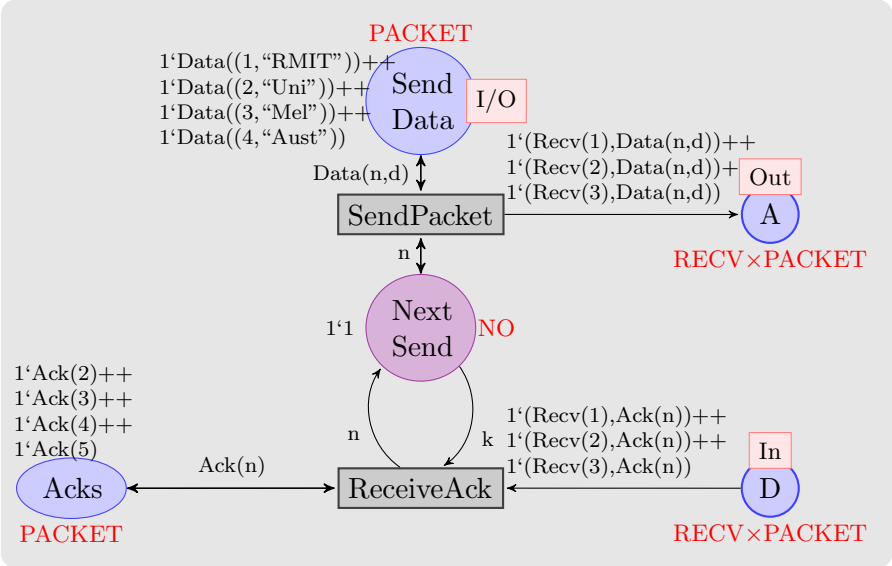


Figure 4.2: Module for the Sender in Fig 4.1

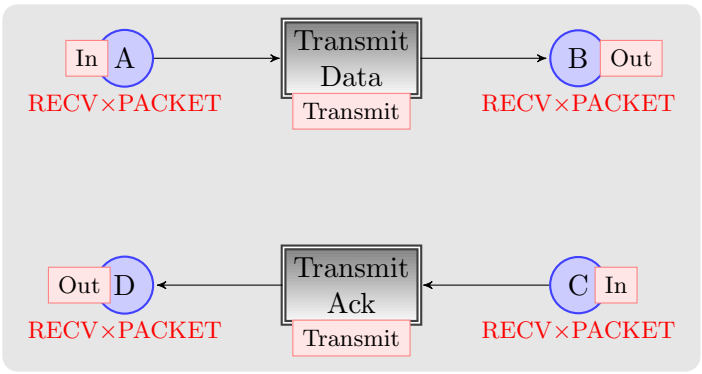


Figure 4.3: Module for the Network in Fig 4.1

the following holds:

$$\forall p \in P_{In}(t) : (E(p,t)\langle b\rangle \subseteq M(p)) \wedge G(t)\langle b\rangle \tag{4.1}$$

where $P_{In}(t)$ is set set of input places of t, $E(p,t)$ is the expression on arc connecting p to t, $M(p)$ denote the set of tokens in p (known as its *marking*) and $G(t)$ denote the guard, which is nothing but a boolean expression attached to t. For instance the transition *ReceiveAck* in Figure 4.2 has two variables n and k and a binding $\langle$n=3,k=2$\rangle$ would enable it only when $2 \in M(Next\ Send)$ and $Ack(3) \in M(D)$. None of the transitions in example net have an

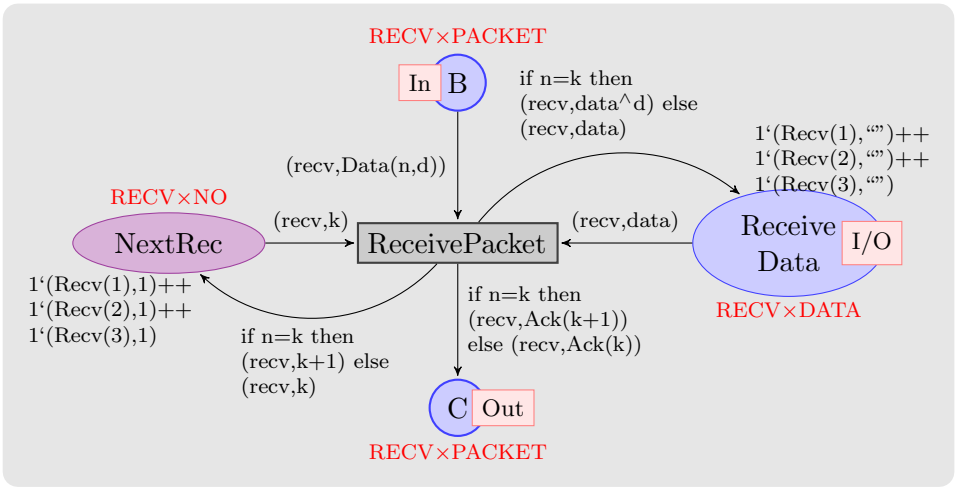


Figure 4.4: Module for the Receiver in Fig 4.1

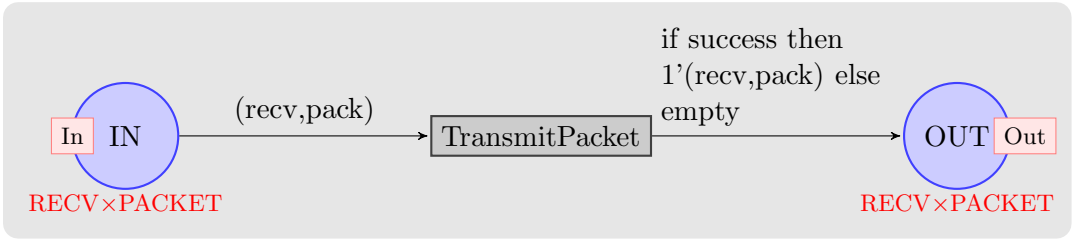


Figure 4.5: Module for the Transmit in Fig 4.3

```
colset NO = int;
colset DATA = string;
colset NOxDATA = product NO * DATA;
colset PACKET = union Data:NOxDATA + Ack:NO;
val NoRecvs = 3;
colset RECV = index Recv with 1..NoRecvs;
colset RECVxPACKET = product RECV * PACKET;
colset RECVxDATA = product RECV * DATA;
colset RECVxNO = product RECV * NO;
var n, k : NO;
var d : DATA;
var pack : PACKET;
```

Figure 4.6: The declarations for modules in Figures 4.1-4.5

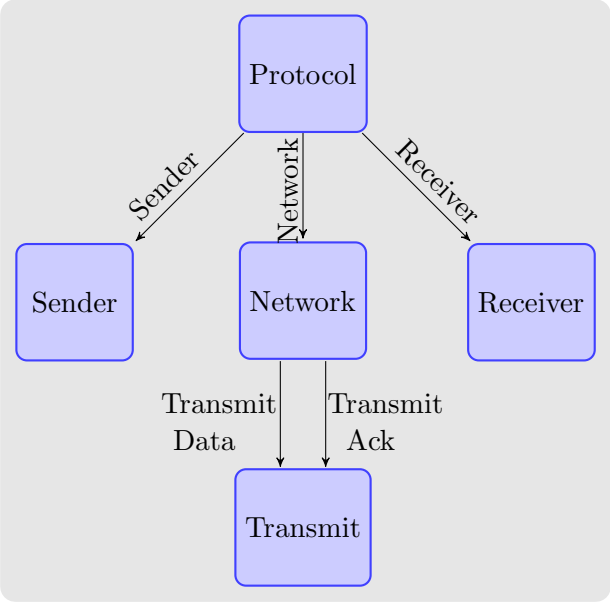


Figure 4.7: Module hierarchy for the example HCPN model.

associated guard condition.

It is worth mentioning that a substitution transition has no associated variables. Consequently it cannot be 'bound' and cannot get 'enabled'. For a particular substitution transition, the tokens in its adjoining places (i.e. sockets) are concurrently available to their associated port in submodule, and vice versa. For instance the socket *Send Data* in Figure 4.1 and its port in Figure 4.2 have the same marking. This allows one or more non-hierarchical transitions in the submodule to get enabled and consume these tokens. After the submodule finishes execution, the left-over tokens at its ports can be claimed back by the sockets.

Reachability Graph for HCPN: Prior to generating the reachability graph for a HCPN model, all its modules are joined to constitute a single large module. This is accomplished by overlapping each port-socket pair, rendering the substitution transitions useless. Each place in the newly constituted module is known as a *compound place*. However, only a subset of these compound places are the outcome of aforesaid port-socket fusion.

Table 4.1 lists the initial marking for each compound place that is constituted out of the example HCPN model. Each port-socket pair in HCPN model would join to constitute a compound place of the new module. Furthermore, a place in HCPN model that is neither a port nor a socket(e.g. *Next Send*) would be a compound place by itself. The compound place corresponding to a place p in HCPN model is denoted as $[p^*]$. Furthermore, $[p^*] \sim_{cp} [q^*]$

Table 4.1: The compound places in HCPN model along with their initial marking

Compound Place	**Initial Marking M_0**
$[SendData^*_{Protocol}] \sim_{cp} [SendData^*_{Sender}]$	$1`Data((1, \text{“}RMIT\text{”}))$ ++ $1`Data((2, \text{“}Uni\text{”}))$ ++ $1`Data((3, \text{“}Mel\text{”}))$ ++ $1`Data((4, \text{“}Aust\text{”}))$
$[A^*_{Protocol}] \sim_{cp} [A^*_{Sender}] \sim_{cp} [A^*_{Network}] \sim_{cp} [IN^*]$	$\emptyset$
$[D^*_{Protocol}] \sim_{cp} [D^*_{Sender}] \sim_{cp} [D^*_{Network}] \sim_{cp} [OUT^*]$	$\emptyset$
$[B^*_{Protocol}] \sim_{cp} [B^*_{Network}] \sim_{cp} [B^*_{Receiver}] \sim_{cp} [OUT^*]$	$\emptyset$
$[C^*_{Protocol}] \sim_{cp} [C^*_{Network}] \sim_{cp} [C^*_{Receiver}] \sim_{cp} [IN^*]$	$\emptyset$
$[ReceiveData^*_{Protocol}] \sim_{cp} [ReceiveData^*_{Receiver}]$	$1`(Recv(1), \text{“”})$ ++ $1`(Recv(2), \text{“”})$ ++ $1`(Recv(3), \text{“”})$
$[NextSend^*]$	$1`1$
$[Acks^*]$	$1`Ack(2)$ ++ $1`Ack(3)$ ++ $1`Ack(4)$ ++ $1`Ack(5)$
$[NextRec^*]$	$1`(Recv(1), 1)$ ++ $1`(Recv(2), 1)$ ++ $1`(Recv(3), 1)$

denotes that p and q belong to same compound place. Any conflict in place names are resolved by including the module name as a subscript. The new module has 8 compound places corresponding to the 8 rows in Table 4.1.

The multiple entries of compound places $[IN^*]$ and $[OUT^*]$ in Table 4.1 corresponds to separate instances of module *Transmit*. While the place *IN* from first instance forms a compound place with A, the same place from other instance forms a compound place with C. Similarly the two instances of OUT form compound places with B and D.

The initial marking constitutes the root node of the reachability graph. The enabled events at this marking can bring in a change in state. The root node has an outgoing edge for each of these enabled marking that leads to a new node. The markings corresponding to these new nodes are then analysed to find the next set of enabled events that would lead to another set of new nodes. The analysis continues until there are no enabled events.

However, if there exists a non-empty sequence of events that causes no net change in a marking M (i.e. $M[e_1\rangle M_1[e_2\rangle M_2 \cdots M_{r-1}[e_r\rangle M : r>0$), the states $\{M, M_1 \cdots, M_{r-1}\}$ would be analysed forever. In order to ensure termination, all the unique marking encountered hitherto are stored and compared with the markings generated hereafter.

4.4 Related Work

All existing solutions for reducing the time requirement for model-checking can be categorised as either of 1) Partial order reduction 2) Symmetry based reduction or 3) Modular state-space generation . We discuss the solutions corresponding to each of these categories and compare them to the proposed technique.

Partial order techniques for HCPN involves determining the *stubborn-sets* and executing only the enabled transitions in each set. A stubborn-set consists of a set of transitions such that a transition outside the set cannot effect their behaviour. Consequently it requires the presence of *concurrent independent transitions* which would lead the system to the same marking irrespective of their order of execution. However, the problem of deciding if a set of transitions is stubborn at a state is at least as hard as the reachability problem [Clarke et al., 2000]. Partial-order techniques for Coloured Petri-nets have been proposed in [Evangelista and Pradat-Peyre, 2006] and [Kristensen and Valmari, 1998] and their worst observed efficiencies are illustrated in Table 4.3.

The symmetry method exploits the presence of any underlying symmetry or symmetrical components in the target system. The symmetric components in such systems exhibit identical behaviour and have identical state graphs. The sub-graphs of these components in the reachability graph of the entire system are usually interchangeable with some permutation of states. Therefore the system reachability graph could be broken into symmetrical graph quotients. One of these graph quotients, when annotated with corresponding permutations, could be enough to verify the properties of the entire system. However, it is difficult to determine a graph quotient whose permutations would produce other graph quotients (known as the *orbit problem* [Clarke et al., 1998; Emerson and Sistla, 1996]). Solving the orbit problem is at least as hard as the *graph isomorphism problem* [Clarke et al., 1998; Köbler et al., 1993] that requires determining if two finite graphs are isomorphic. Symmetry method for coloured Petri-nets was proposed in [Elgaard, 2002]. The performance of symmetry based algorithms depend on the extent of symmetry in the target system.

Modular state-space generation involves generating the reachability graph of each module independently and then composing them to generate the reachability graph for the entire system. The proposed algorithm is based on this technique. Although the algorithm proposed in [Christensen and Petrucci, 1995] is also based on this technique, the semantics of 'transition fusion' used in this algorithm is no longer defined in CPN tools and is best avoided. The solutions based on this technique only necessitate a hierarchical modeling language to be

used for system representation. Consequently it is not necessary to linger in identifying the stubborn sets or the symmetrical markings. Furthermore, it is possible to reduce the time overhead by generating the reachability graph for each module in parallel.

Table 4.2: A comparison of the categories of existing solutions

Criteria	**Categories of existing solutions**		
	Partial-Order	*Symmetry*	*Modular*
Requires	Stubborn-sets	Symmetry	Modules
Mechanism	Determine concurrent independent transitions	Determine representative states	Determine and merge the reachability graphs of different modules
Complexity	NP	NP	P
Estimation	Heuristic	Heuristic	-

Table 4.2 compares the broad categories of existing solutions. The proposed solution is based on modular technique which has a polynomial solution.

Table 4.3 compares the proposed technique with [Evangelista and Pradat-Peyre, 2006] and [Kristensen and Valmari, 1998] that are based on partial order techniques. While the former offers a reduction of 50.82% in worst case scenario, the latter offers a meagre 10.7% reduction. These are significantly less than 86% reduction offered by the proposed technique in all cases. Furthermore the prerequisite of proposed solution (i.e. a hierarchical model) is less stringent as compared to these techniques wherein a stubborn set is mandatory. The reduction offered by Symmetrical reduction techniques largely depend on the extent of symmetry in the model under deliberation.

Table 4.3: A comparison of the existing solutions

Method	**Run-Time**	**Number of Markings**
No Algorithm	100%	Any
[Evangelista and Pradat-Peyre, 2006]	49.18%	45780
[Kristensen and Valmari, 1998]	89.3%	25
Proposed technique	14%	25000

4.5 Proposed Technique for Time Efficient State-Space Analysis

In this section, we propose a technique for time-efficient state-space analysis of HCPN models. It can be extended for other modeling languages that define a notion of hierarchy.

As observed previously, the modules of a HCPN model are joined prior to generating the reachability graph. This in effect amounts to flattening the model and analysing its equivalent non-hierarchical counterpart. Despite this being a legitimate solution, it lacks concurrency and parallelism. Our technique installs these features by exploring the HCPN modules in parallel instead of joining them together. The existing dependencies on a module are represented using parameters. Such a setup provide the envisioned reduction in time requirement.

Figure 4.8 illustrates the concurrent exploration of the HCPN modules. However, as depicted in Figure 4.7, the *Protocol* module is dependent on three of its submodules. Consequently it needs to access the reachability graph (RG) for each of *Sender*, *Network* and *Receiver* when generating its own reachability graph. Similarly the *Network* module requires the reachability graph for *Transmit* module. While the vertical lines in Figure 4.8 denote the execution of a module, the horizontal lines denote a module trying to access the reachability graph for one of its submodules. In case the reachability graph for a submodule is available, it is returned at once. Otherwise the module needs to wait until it becomes available. For instance, when *Protocol* tries to fetch the reachability graph for *Sender* at time T_p2, the latter has not yet finished generating it. Consequently it waits till time T_p3 when the required reachability graph is available. However it is not required to wait when it tries to fetch the reachability graph for *Network* or *Receiver* as these modules have finished execution by then and the reachability graph is available right away.

Algorithm 7 lists the steps for executing all modules concurrently. A new thread is created for each module in S and all the threads explore the corresponding module concurrently. The steps for exploring a module is illustrated later in Algorithm 9. The next section deliberate using parameters to represent dependency between modules.

Algorithm 7: ConcurrentRG(S)

Data: The set of modules S

Result: Each module is explored concurrently

```
1 foreach module m ∈S do
2 |   Thread t=new Thread(new Explore(m));
3 |   t.start();
4 end
```

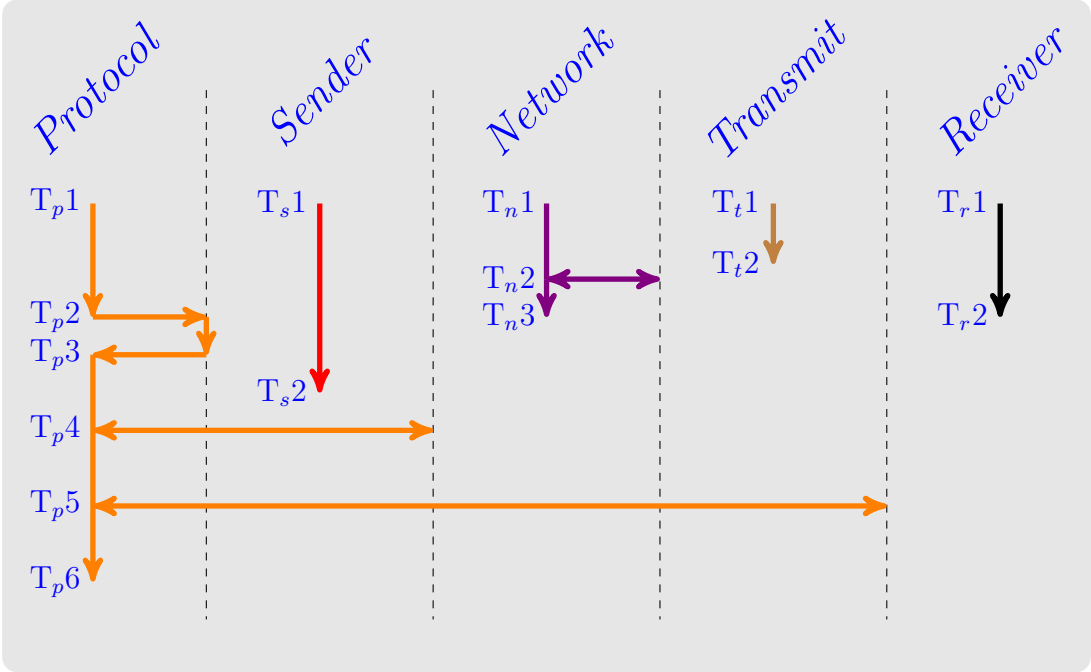


Figure 4.8: The order of access for modules in the example HCPN model.

4.5.1 Access-Table and Parametrised Reachability Graph

In this section we introduce access-tables and parameters for managing the dependencies on a module. Figure 4.7 depicts the existing dependencies for example HCPN.

As observed previously, a substitution transition executes by furnishing the associated submodule with tokens at each of its input places. The submodule receives these tokens at its ports that are tagged 'In' and executes all the enabled transitions. Thereafter the tokens at its ports tagged 'Out' are made available to the substitution transition at its output places. On furnishing different sets of tokens to the submodule, the substitution transition might expect different sets of tokens returned into its output places. From the perspective of the substitution transition, the submodule is a black box which when supplied with a particular set of tokens always returns another set of it. If a table contains all the possible inputs that this submodule can accept and the result produced in each case, we can replace the submodule with this table. Considering the deterministic nature of a submodule, the substitution transition would never know that the results are being fetched from a table. Since the result of executing a module is available without actually executing it, the scheme would render a massive reduction in delay. This table is known as *access-table* and the steps

in generating it for a module are listed later as an algorithm.

An access-table has one column for each port and a set of additional columns to check the validity of conditions. Each entry in the access-table consists of 1) A minimum set of tokens required in input ports, 2) The possible values of listed conditions (if applicable) and 3) The set of tokens at the output ports under each condition. Table 4.4 shows the access-table for the *Sender* module. The first entry requires port *Send Data* to at least contain a token *Data(n,d)*, where n and d are parameters. The use of parameters helps in reducing the number of rows in access-table by allowing a range of token values. The column for conditions lists all possible results when evaluating the specified condition *n=v?*. The variable v stores the value of token at place *Next Send*. Finally, the marking of output ports for each of these conditions are listed. It is worth mentioning that we treat I/O ports as a combination of input and output ports. Consequently they have one column under each of these categories. The superscript of a token indicates if it is being added or removed ('+'=add, '-'=remove).

If a substitution transition uses an access-table instead of actually executing its sub-module, any required changes in marking of non-port places would be omitted. This is because the access-table only accounts for the port places of a submodule. For instance the access-table does not account for change in marking of place *Next Send* when module *Sender* executes with token *Ack(m)* in place *D* (2^{nd} row of Table 4.4). Nevertheless, this could steer the model into an incorrect marking to produce an incorrect reachability graph (RG). Consequently, each row of the access-table is associated with a *parametrised reachability graph* (PRG). A PRG is generated by initialising the input port places of a submodule with tokens in the corresponding row of access-table and exploring it. Figures 4.9 and 4.10 illustrate the parametrised reachability graphs corresponding to the two rows of its access-table. The parametrised reachability graphs corresponding to the 2^{nd} row now accounts for the change in marking of the non-port place *Next Send*.

When generating the reachability graphs for *Protocol*, the substitution transition *Sender* is encountered. Supposing that the access-table and parametrised reachability graphs for *Sender* have already been generated, all that is required is to determine the most appropriate entry in the access-table. Considering that place *Send Data* has three tokens while *D* has none, the first row in Table 4.4 turns out to be the obvious choice. The values assigned to parameters n and d by each of these three tokens are shown in Table 4.5. The value of variable v is '1', owing to the token *1'1* in place *Next Send*. Consequently, only the token *1'Data(1,"RMIT")* satisfies the condition *n=v?* and the only token added to place *A* is *1'Data(1,"RMIT")*. This being an output port, the token also appears in an output place

Table 4.4: The access-table for Sender

Input Ports		**Conditions**	**Output Ports**	
Send Data	**D**	**n=v?**	**A**	**Send Data**
$Data(n,d)^-$	empty	YES	1‘$(Recv(1),Data(n,d))^+$ ++ 1‘$(Recv(2),Data(n,d))^+$ ++ 1‘$(Recv(3),Data(n,d))^+$	$Data(n,d)^+$
		NO	empty	$Data(n,d)^+$
empty	1‘$(Recv(1),Ack(m))^-$ ++ 1‘$(Recv(2),Ack(m))^-$ ++ 1‘$(Recv(3),Ack(m))^-$	-	empty	empty

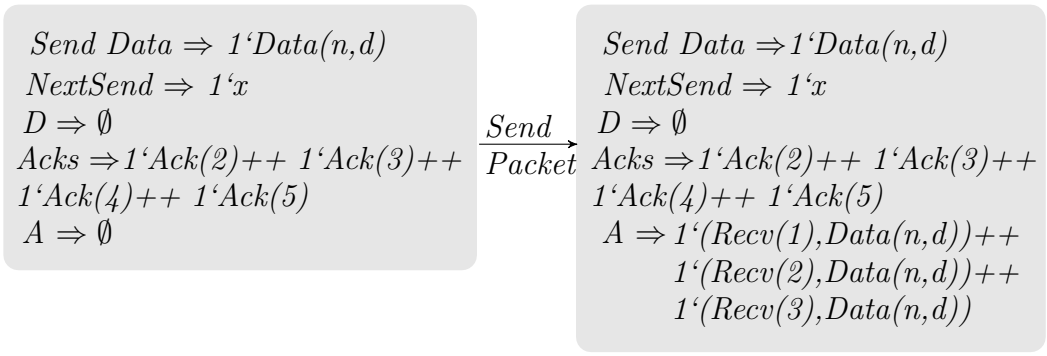


Figure 4.9: Reachability Graph for first row in Table 4.4.

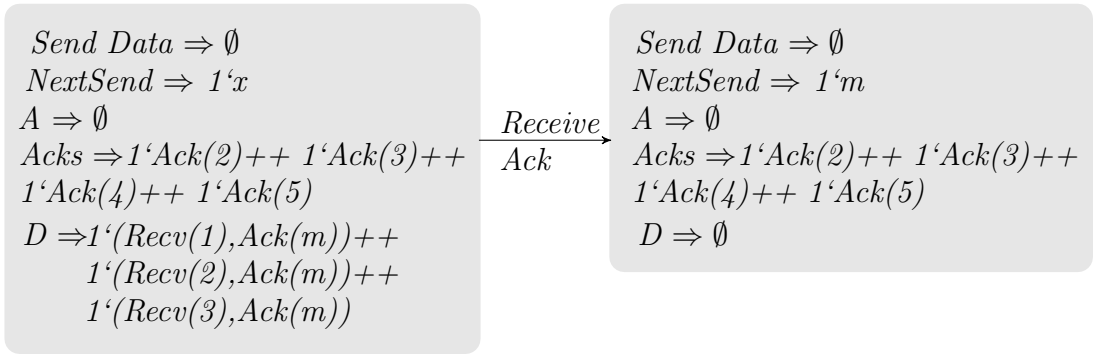


Figure 4.10: Reachability Graph for second row in Table 4.4.

Table 4.5: The values assigned to parameters by tokens

Token	n	d
Data(1,"RMIT")	1	"RMIT"
Data(2,"Uni")	2	"Uni"
Data(3,"Mel")	3	"Mel"
Data(4,"Aust")	4	"Aust"

of substitution transition *Sender* where it is available for the module *Network*. Finally, the parametrised reachability graph is checked to update the markings of any non-port places of the module *Sender*.

The above discussion underlines the role played by access-tables and parametrised reachability graph in concurrent state-space analysis. Each submodule generates its own access-table and associated parametrised reachability graphs in parallel as shown in Figure 4.8. As observed previously, a supermodule needs to wait if its submodule is still in the process of generating them. The next section further illustrates the importance of access-tables and explains the steps in constructing them.

4.5.2 Exploring a Module

When generating the access-table or parametrised reachability graph of a module, the most challenging aspect is to determine the set of initial marking for its input port-places that enable the module. Each distinct initial marking that enables the module would add a new row into the access-table and create an associated parametrised reachability graph. If a substitution transition queries an access-table with the tokens at its input places and does not find a match, it concludes that the associated submodule is not enabled for the particular marking. Therefore omitting a marking that enables the module would lead to potential error.

The proposed algorithm determines the initial markings by identifying the set of bindings that enable the output transitions of input port-places. This follows from the fact that enabling any of these transitions enables the module. For each of these bindings, the marking of input port-places could be established by evaluating the corresponding arc expressions. Considering that each of these markings would enable at least one transition of the module, all possible combinations of these markings would return the required exhaustive set of initial markings. We now discuss the underlying strategy before proposing the algorithm.

Each arc connecting a transition t to its adjoining places have an associated arc expression.

In addition, t might also have an associated expression as guard condition. The variables constituting these expressions are known as *free-variables (FV)* [Kristensen and Christensen, 2004]. Accordingly, the set of free-variables for transition t is given by

$$FV(t) = \bigcup_{p \in P_{in}(t)} FV(E(p,t)) \cup \bigcup_{p \in P_{out}(t)} FV(E(t,p)) \cup FV(G(t)) \tag{4.2}$$

where (1) E(p,t) denote the arc expressions on input arcs of t (2) E(t,p) denote the arc expressions on output arcs of t and (3) G(t) denote the guard condition for t. The set of FV for the transition *ReceivePacket* appearing in Figure 4.4 is expressed as

$$FV(ReceivePacket) = \{n, recv, k, data, d\} \tag{4.3}$$

It should be noted that the scope of a CPN FV is limited by a transition, i.e. a FV appearing in multiple arc expressions or guard for a transition t is the same variable. For instance the variable *recv* appearing in each input arc expression of *ReceivePacket* is the same.

When generating the reachability graph for a module, it is necessary to determine the set of enabled transitions at every step. In order to determine if a transition is enabled, all the FVs associated with it need to be bound to values and the corresponding arc expression be evaluated. This was observed in Section 4.3.

Definition 7 *Binding is the process of assigning values to each FV associated with a transition. It is written in the form* $b=<v_1=c_1,\ v_2=c_2, \cdots\ v_n=c_n>$ *where* $FV(t)=\{v_i | i \in 1 \cdots n\}$ *and* c_i *is the value bound to* v_i*.*

The result on evaluating an arc expression e for binding $b=<v_1=c_1,\ v_2=c_2, \cdots\ v_n=c_n>$ is defined as

$$e < b >= (fn(v_1, \cdots, v_n) \Rightarrow e)(c_1, \cdots, c_n) \tag{4.4}$$

The place *NextRec* in Figure 4.4 contains three tokens. In order to ensure that *ReceivePacket* is enabled, the expression on arc connecting *NextRec* to *ReceivePacket* must evaluate to one

of these three tokens. Consequently the possible bindings should be

$$b_1 =< recv = Recv(1), k = 1 > \tag{4.5}$$

$$b_2 =< recv = Recv(2), k = 1 > \tag{4.6}$$

$$b_3 =< recv = Recv(3), k = 1 > \tag{4.7}$$

On evaluating the expression *(recv,k)* with b1, b_2 and b_3, the resultant arc-expressions obtained should culminate in removal of either of the three tokens in *NextRec*.

However, out of the five FVs of *ReceivePacket* shown in equation 4.3, only two have been bound in either of b_1, b_2 or b_3. Other variables will be bound similarly by matching the tokens at input places with the expressions appearing on input arcs. This requirement is captured by the *pattern binding basis* [Kristensen and Christensen, 2004].

Definition 8 *A pattern binding basis PBB(t) for a transition t is a set of input arc expressions of t satisfying:*

1. *FV(t)=*$\bigcup_{E(p,t)\in PBB(t)} FV(E(p,t))$
2. $\forall$*E(p,t)* $\in$ *PBB(t): PATTERN(E(p,t))*

Definition 9 *A pattern is an expression constituted of constructors, identifiers and constants.*

The first item ensures that the FVs of a transition t appear in at least one of the expressions in *PBB(t)*. The second item ensures that each expression in *PBB(t)* is a pattern (Definition 9). The *PBB* for the transition *ReceivePacket* in *Receiver* is

$$PBB(ReceivePacket) = \{(recv, k), (recv, Data(n, d)), (recv, data)\} \tag{4.8}$$

The bindings b_1, b_2 and b_3 are only *partial-bindings* as they assign values to a subset of variables in FV(ReceivePacket). In general, matching the tokens at an input place to the expression on the arc connecting it to a transition t would normally bind only a subset of free variables fv$\subseteq$FV(t) of t. The partial-bindings for transition *ReceivePacket* that are obtained

from bindings in equations 4.5, 4.6 and 4.7 are

$$pb^1_{ReceivePacket} =< recv = Recv(1), k = 1, n =\perp, d =\perp, data =\perp > \tag{4.9}$$

$$pb^2_{ReceivePacket} =< recv = Recv(2), k = 1, n =\perp, d =\perp, data =\perp > \tag{4.10}$$

$$pb^3_{ReceivePacket} =< recv = Recv(3), k = 1, n =\perp, d =\perp, data =\perp > \tag{4.11}$$

where $\perp$ denotes that a variable is not bound to a value.

Since the remaining two input places of *ReceivePacket* are port-places, they do not have a fixed marking. Their binding is determined by the input places of the corresponding substitution transition and consequently they cannot be used to bind free variables. In order to enable transition *ReceivePacket*, we add tokens to each of these input ports based on the following rules:

1. The tokens added to a port place is determined by evaluating the expression on the arc connecting it to the transition. The partial-bindings of the transition are used in evaluating these arc-expressions.

2. When evaluating the arc expression in the previous step, each FV that is unbound in partial binding is assigned a unique parameter.

3. The colour of the token added must be same as that of the containing port-place.

Using these rules, the tokens added to the input port-places of *ReceivePacket* are

$$B =1`(Recv(1), Data(n_p, d_p)) + +1`(Recv(2), Data(n_p, d_p)) + + 1`(Recv(3), Data(n_p, d_p)) \tag{4.12}$$

$$Receive\ \ Data =1`(Recv(1), data_p) + +1`(Recv(2), data_p) + +1`(Recv(3), data_p) \tag{4.13}$$

The expression on arc connecting place B to *ReceivePacket* is *(recv,Data(n,d))*. Because neither n nor d were assigned a value in the partial-bindings $pb^1_{ReceivePacket}$, $pb^2_{ReceivePacket}$ or $pb^3_{ReceivePacket}$(equations 4.9-4.11), they are now assigned parameters n_p and d_p. For the same reason, FV *data* is assigned parameter $data_p$. The bindings after adding these tokens

Table 4.6: The tokens removed by ReceivePacket from its input places for each binding

Binding	**Input Places**		
	B	**NextRec**	**Receive Data**
b $^{1}_{ReceivePacket}$	1‘(Recv(1),Data(n_p, d_p))	1‘(Recv(1), k)	1‘(Recv(1), $data_p$)
b $^{2}_{ReceivePacket}$	1‘(Recv(2),Data(n_p, d_p))	1‘(Recv(2), k)	1‘(Recv(2), $data_p$)
b $^{3}_{ReceivePacket}$	1‘(Recv(3),Data(n_p, d_p))	1‘(Recv(3), k)	1‘(Recv(3), $data_p$)

are as follows:

$$b^1_{ReceivePacket} =< recv = Recv(1), k = 2, n = n_p, d = d_p, data = data_p > \quad (4.14)$$

$$b^2_{ReceivePacket} =< recv = Recv(2), k = 1, n = n_p, d = d_p, data = data_p > \quad (4.15)$$

$$b^3_{ReceivePacket} =< recv = Recv(3), k = 2, n = n_p, d = d_p, data = data_p > \quad (4.16)$$

When the transition *ReceivePacket* fires, it removes the token from input places and moves tokens to output places. The tokens removed and added depends on the binding for which the transition fires. The tokens removed for each binding are shown in Table 4.6. The entry for each binding in Table 4.6 maps to a row in the corresponding access-table. However, only the information for port-places B and *Receive Data* are used, as access-tables do not have columns for other places. The value of k in each entry is fetched from the corresponding binding.

The output arcs of *ReceivePacket* have a conditional expression ‘*if* n=k’ and the tokens added to output places depend on the boolean result obtained on evaluating this expression. Accordingly, there is an additional column in the access-table to check this condition that contains all possible results of evaluating the expression. The tokens added to each output place for each possible outcome of conditional expression are shown in Table 4.7. Each entry in Table 4.7 corresponds to a row in the corresponding access-table. However, only the information for port-places B and *Receive Data* are used, as access-tables do not have columns for other places. The value of k in each entry is fetched from the corresponding binding. The access-table for *Receiver* can be constituted from Tables 4.6 and 4.7 by joining the entries for identical bindings and removing the excess columns. The obtained access-table is shown in Table 4.8 .

Since the module *Receiver* contains a single transition, its access tables could be constructed using tables that contained information about the tokens it adds and removes. However, if the module had additional transitions, we would have required similar tables for

Table 4.7: The tokens added by ReceivePacket to its output places for each binding

Binding	**Condition**	**Output Places**		
	n=k?	**C**	**NextRec**	**Receive Data**
b $^{1}_{ReceivePacket}$	YES	1‘(Recv(1), Ack(k+1))	1‘(Recv(1), k+1)	1‘(Recv(1), $data_p$∧d))
	NO	1‘(Recv(1), Ack(k)	1‘(Recv(1), k)	1‘(Recv(1), $data_p$)
b $^{2}_{ReceivePacket}$	YES	1‘(Recv(2), Ack(k+1))	1‘(Recv(2), k+1)	1‘(Recv(2), $data_p$∧d))
	NO	1‘(Recv(2), Ack(k)	1‘(Recv(2), k)	1‘(Recv(2), $data_p$)
b $^{3}_{ReceivePacket}$	YES	1‘(Recv(3), Ack(k+1))	1‘(Recv(3), k+1)	1‘(Recv(3), $data_p$∧d))
	NO	1‘(Recv(3), Ack(k)	1‘(Recv(3), k)	1‘(Recv(3), $data_p$)

Table 4.8: The access-table of Receive from Tables 4.6 and 4.7

Input Ports		**Condition**	**Output Ports**	
B	**Receive Data**	**n=k?**	**C**	**Receive Data**
1‘(Recv(1),Data(n_p, d_p))$^-$	1‘(Recv(1), $data_p$)$^-$	YES	1‘(Recv(1), Ack(k+1))$^+$	1‘(Recv(1), $data_p$∧d))$^+$
		NO	1‘(Recv(1), Ack(k))$^+$	1‘(Recv(1), $data_p$))$^+$
1‘(Recv(2),Data(n_p, d_p))$^-$	1‘(Recv(2), $data_p$)$^-$	YES	1‘(Recv(2), Ack(k+1))$^+$	1‘(Recv(2), $data_p$∧d))$^+$
		NO	1‘(Recv(2), Ack(k))	1‘(Recv(2), $data_p$))
1‘(Recv(3),Data(n_p, d_p))$^-$	1‘(Recv(3), $data_p$)$^-$	YES	1‘(Recv(3), Ack(k+1))$^+$	1‘(Recv(3), $data_p$∧d))$^+$
		NO	1‘(Recv(3), Ack(k))$^+$	1‘(Recv(3), $data_p$))$^+$

Table 4.9: The access-table for Transmit

Input Ports	**Conditions**	**Output Ports**
IN	**Success?**	**OUT**
1‘(Recv(k),Data(n,d))$^-$	YES	1‘(Recv(k),Data(n,d))$^+$
	NO	empty
1‘(Recv(k),Ack(n))$^-$	YES	1‘(Recv(k),Ack(n))$^+$
	NO	empty

all those transitions in order to generate the access-table.

Often it is only the output arc expressions that contain the conditional expressions. The expressions on input arcs are almost always very simple. Hence the column for ‘Condition’ is missing from the table that illustrates the tokens removed from input places.

4.5.3 Access-table & Parametrised reachability graph for a Super-Module

In the previous section we discussed the role played by access-tables and parametrised reachability graphs in concurrent state-space analysis. However, the discussion was limited to modules that do not have any further submodules. In this section, we discuss producing the parametrised reachability graph and access-tables for such super-modules.

For instance the module *Network* has a submodule *Transmit* as illustrated in Figure 4.7. Accordingly, as shown in Figure 4.8, it would need the access-table for *Transmit* in order to produce its own access-table. Considering that *Transmit* does not have a submodule, its access-table can be created based on the aforementioned discussion and is shown in Table 4.9. Note that the parameters used in this table are of simple types (e.g. INT, STRING etc.) instead of user defined types (i.e. RECV, PACKET or their product) that appear in arc-expression. This requires modifying the arc-expression before applying the three rules proposed in the previous section for adding tokens to input ports. Using parameters of simple types throughout a model prevents any ambiguities in assigning them values. This is further illustrated towards the end of this section.

Network has two input-ports, *A* and *C*. Considering that these are input places of substitution transitions *Transmit Data* and *Transmit Ack*, there are no free-variables or arc-expressions to evaluate and determine the minimum number of tokens to be added into these ports in order to enable the module *Network*. Consequently the access-table for *Transmit* module (corresponding to these substitution transitions) is used to ascertain the enabling tokens to be inserted into these ports.

Table 4.10: The access-table for Network

Input Ports		**Conditions**	**Output Ports**	
A	**C**	**Success?**	**B**	**D**
1‘(Recv(k), Data(n,d))$^-$	empty	YES	1‘(Recv(k), Data(n,d))$^+$	empty
		NO	empty	empty
1‘(Recv(k), Ack(n))$^-$	empty	YES	1‘(Recv(k), Ack(n))$^+$	empty
		NO	empty	empty
empty	1‘(Recv(k), Data(n,d))$^-$	YES	empty	1‘(Recv(k), Data(n,d))$^+$
		NO	empty	empty
empty	1‘(Recv(k), Ack(n))$^-$	YES	empty	1‘(Recv(k), Ack(n))$^+$
		NO	empty	empty

From Table 4.9, it can be inferred that the presence of either *(Recv(k),Data(n,d))* or *(Recv(k),Ack(n))* in place *IN* enables the *Transmit* module. Considering that

1. Enabling of *Transmit* module enables the substitution transitions, which in turn enable the module *Network*, and

2. The port *IN* of *Transmit* maps to socket-places *A* and *C* of *Network*,

it can be deduced that adding either of these tokens to places *A* or *C* would enable the *Network* module. Accordingly the access-table for *Network* contains an entry for adding each of these tokens into a socket-place and is shown in Figure 4.10.

In general, prior to generating the access-table for a super-module, the tokens that enable each of its submodule need to be determined from their access-tables. The columns of an access-table convey the tokens that need to be added into each input port-place in order to enable the submodule. The supermodule has one or more substitution transitions corresponding to each of these submodules, and based on the port-socket mapping, its sockets are populated with the enabling tokens from their ports. A sockets could either be

1. an input port-place of the supermodule. In that case, the tokens in it add entries into the access-table of supermodule.

2. a non-input port place of the supermodule. In that case, the set of tokens in input-port places needs to be determined that would produce the enabling tokens in the socket.

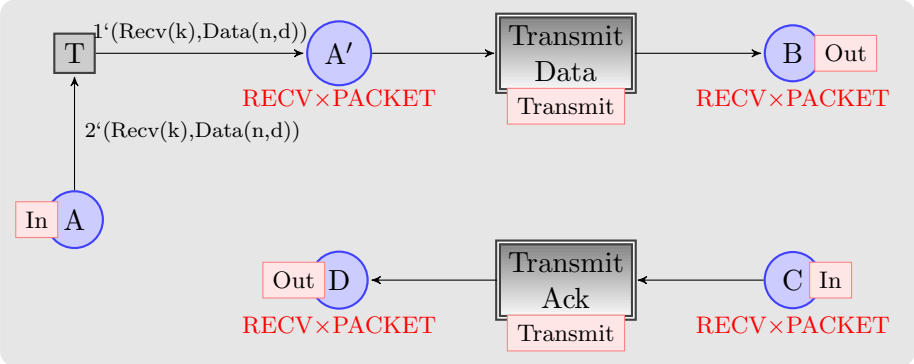


Figure 4.11: Module for the Network in Fig 4.1

Table 4.11: Tokens in input port-places that produce the enabling tokens in socket

A′	**A**	**C**
1‘(Recv(k), Data(n,d))	2‘(Recv(k), Data(n,d))	none
1‘(Recv(k), Ack(n))	none	none

This is done by backtracking. Furthermore, the set of tokens in input port-places that produce the enabling tokens add entries into the access-table for supermodule.

The first case was observed for *Network* module where the socket places A and C were also the input port-places. In order to demonstrate the second case, the *Network* module is modified as shown in Figure 4.11. The socket A' is not an input port-place of *Network*. Therefore the tokens in input port-places need to be determined in order to generate the access table. The enabling tokens that could be added to A' are *1‘(Recv(k), Data(n,d))* and *1‘(Recv(k), Ack(n))*. The corresponding tokens on input port-places A and C are shown in Table 4.11. While two instances of *(Recv(k), Data(n,d))* in A produce the first enabling token in A', the other enabling token cannot be produced for any combination of tokens in input port-places. Accordingly the access-table for modified *Network* module does not have a entry for the second enabling token as shown in Table 4.12. Furthermore, the entry for the first enabling token now requires 2 instances of *(Recv(k), Data(n,d))* in A.

A comparison of Table 4.10 with Tables 4.4 and 4.8 exemplify the advantages in using simple parameters. The columns A, B, C and D in all these tables contain tokens that are compatible (i.e. of the same data-type) and similar. Consequently when the tokens move between modules, a simple assignment of enabling tokens would determine the required result.

We now propose the algorithms for generating the access-table and parametrised reachability graphs of a module.

Table 4.12: The access-table for modified Network module

Input Ports		**Conditions**	**Output Ports**	
A	**C**	**Success?**	**B**	**D**
2‘(Recv(k), Data(n,d))$^-$	empty	YES	1‘(Recv(k), Data(n,d))$^+$	empty
		NO	empty	empty
empty	1‘(Recv(k), Data(n,d))$^-$	YES	empty	1‘(Recv(k), Data(n,d))$^+$
		NO	empty	empty
empty	1‘(Recv(k), Ack(n))$^-$	YES	empty	1‘(Recv(k), Ack(n))$^+$
		NO	empty	empty

4.5.4 Algorithms for Generating Access-Tables and parametrised reachability graphs

The algorithm for generating the reachability graph of a module is now proposed. Before proposing the algorithm, we define a few terms used in the algorithm.

Definition 10 *Partial bindings pb_1 and pb_2 for a transition t are said to be compatible (denoted as Compatible(pb_1, pb_2)) if they have consistent values bound to their variables,*

$$i.e. \qquad \forall v \in FV(t) : pb_1(v) \neq \perp \wedge pb_2(v) \neq \perp \Rightarrow pb_1(v) = pb_2(v)$$

Definition 11 *If pb_1 and pb_2 are two partial bindings such that Compatible(pb_1, pb_2), they can be combined (denoted as Combine(pb_1, pb_2)) to obtain another partial binding pb where*

$$pb(v) = \left\{ \begin{array}{lcl} pb_1(v) & : & pb_1(v) \neq \perp \\ pb_2(v) & : & pb_2(v) \neq \perp \\ \perp & : & otherwise \end{array} \right\} \tag{4.17}$$

Definition 12 *Merging is the process of combining two sets of partial bindings such that for any two sets B_1 and B_2*

$$\begin{aligned} Merge(B_1, B_2) = &\{Combine(pb_1, pb_2) | \exists (pb_1, pb_2) \in B_1 \times B_2 : \\ &Compatible(pb_1, pb_2)\} \end{aligned} \tag{4.18}$$

Definition 13 *PB(exp,tok_{val}) is the partial-binding obtained by matching an expression exp with a token value tok_{val}. If they do not match, PB(exp, tok_{val})=$\perp$.*

Algorithm 8 lists the steps for determining the enabled bindings for a particular transition. It is a modified version of the algorithm proposed in [Kristensen and Christensen, 2004] and can now handle port places.

The loop in steps 2-12 fetches each pattern E(p,t) from PBB(t) and computes the associated partial binding C' by comparing the pattern with the tokens in corresponding input place p. However, as shown in step 3, the pattern is not compared if p is a port place. This follows from the fact that a port can have disparate markings obtained from its associated sockets. Before considering the next pattern, this partial binding is then merged with the current partial binding C.

The set of partial bindings obtained is then processed in a loop (steps 13-19) wherein each unassigned free-variables in a binding is attached to a unique parameter. As described later, this allows ports to have generic tokens that can be assigned values from any token supplied by a socket. The value of a FV v in a binding b is contained in $b(v)$ (steps 15-16).

Finally the bindings in C are subjected to the guard condition in step 20 in order to filter out any bindings in conflict. Furthermore if the patterns in PBB(t) does not include an input arc-expression E(p,t):p$\notin$P$_{port}$, the corresponding input-place p is checked (step 21) to ensure it contains the minimum number of required tokens.

Algorithm 9 lists the steps for generating all possible initial markings that enable a module. The single most important thing to do before exploring a model is to populate its input-ports with generic tokens. In this pursuit, the set of output transitions for all port places are determined (in step 2) and their enabled bindings are ascertained (steps 4-7) using algorithm 8. Thereafter the marking for each port place p is modified to contain additional tokens (steps 8-10). Each additional token added to a port corresponds to the result of evaluating its arc expression with an enabled binding.

In order to generate all possible parametrised reachability graphs for a module, all legitimate combinations of initial markings are produced (steps 12-33) and sent to Algorithm 10 for processing (step 19). Prior to generating these combinations, the loop in steps 13-17 store each input port $p \in P^m_{inport}$ in an array $ports[]$ (step 14), while the number of tokens in it is stored in $tokenCount[]$ (step 15). Furthermore, the number of possible initial markings for a port is stored in $activeTokens[]$. A port with x tokens can accept either of $0, 1, 2, \cdots, x$ tokens from a socket initially. The number of ways it can accept these many tokens is

$$ {}^xC_0 +^x C_1 +^x C_2 + \cdots +^x C_x = 2^x \tag{4.19} $$

Algorithm 8: GetBindings(PBB(t))

Data: PBB(t)
Result: The set of enabled bindings

```
1  C ← ∅
2  foreach E(p,t) ∈PBB(t) do
3  |  if p ∈P_port then continue
4  |  C' ← ∅
5  |  foreach c∈M(p) do
6  |  |  b' ← PB(E(p,t),c)
7  |  |  if b' ≠⊥ then
8  |  |  |  C' ← C' ⋃{b'}
9  |  |  end
10 |  end
11 |  C ← Merge(C,C')
12 end
13 foreach b ∈C do
14 |  foreach v ∈FV(t) do
15 |  |  if b(v)=⊥ then
16 |  |  |  b(v)←c_param;
17 |  |  end
18 |  end
19 end
20 C ←{b ∈ C|G(t)⟨b⟩}
21 C ←{b ∈ C|∀p ∈ P_in(t) : p ∉ P_port ∧ E(p,t)⟨b⟩ ⊆ M(p)}
22 return C
```

Consequently $activeTokens[]$ is defined as $2^{tokenCount[]}$ in step 16.

Thereafter the algorithm executes a loop (steps 18-33) wherein M_0 is assigned with all possible combinations of tokens at each of its port places. Considering that the port places in M_0 were initialised with all possible tokens that enable one or more bindings (steps 8-10), the required combinations are obtained by successive removal of tokens from these places starting with the lowest index of array $ports[]$. A port place with x tokens are assumed to have them numbered $1, 2, \cdots, x$. At each step, the value of $activeTokens[]$ for lowest index of $ports$ is decreased (step 22) and the tokens in it are selected using a function $onlyTokens()$ (step 23). This functions selects the set of tokens numbered $\{n_1, n_2, \cdots n_k\}$ out of all tokens in $ports[j]$ such that

$$2^{n_1} + 2^{n_2} + \cdots + 2^{n_k} = activeTokens[j] \tag{4.20}$$

The function *onlyTokens* is computationally linear because it simply involves checking the value of n^{th} bit in $activeTokens[j]$. If it is 1, the token numbered n is included in the marking. The function is listed in Algorithm 11. The function x>>i returns i^{th} bit of x. Step 3 always checks the first bit of $activeTokens[j]$ which is later eliminated in step 6 using an integer division. This operation recursively reduces the value of $activeTokens[j]$ and thereby ensures that the algorithm eventually terminates.

Returning to Algorithm 9, when the value of $activeTokens[]$ for lowest index of $ports[]$ becomes 0, its value for higher indices are polled until a non-zero value is found (step 25). The value of $activeTokens[]$ at this index is decreased (steps 27-28) and its value is restored at all lower indices (steps 29-32) to establish the next possible combination. This is continued till the value of $activeTokens$ for the highest index becomes 0, indicating that none of the port places have any tokens (step 18). Each initial marking obtained is sent to algorithm 10 for processing (step 19).

Algorithm 10 lists the steps for generating the parametrised reachability graph corresponding to an initial marking. It also adds new entries into the access-table corresponding to the module.

When an initial marking is passed into the algorithm, the tokens at each input port place is written into the access-table as a new entry (step 1). Thereafter the sets $Unprocessed$ and $Nodes$ are initialised as shown in steps 2-3. In addition to the initial marking, each entry in $Unprocessed$ contains a field to store the conditions that must be satisfied to reach the marking. This condition is the specific value that needs to be assigned to a subset of parameters to reach a marking. Since there is no condition to reach the initial marking, it is left empty in step 2.

The algorithm executes a loop (steps 4-30) to process the markings in *Unprocessed* until it is empty. At any marking, there could be two possible cases:

1. A transition does not require a parameter to have a specific value in order to get enabled. Furthermore, the result of execution of the transition does not depend on any of the parameters. This case is illustrated using transition *case1* in Figure 4.12 and is handled by the algorithm in steps 6-14.

2. The transition requires one or more parameters to be bound to specific values either to get enabled or to determine the result of its execution. This case is illustrated using transition *case2* in Figure 4.12 and is handled by the algorithm in steps 15-26.

Algorithm 9: Explore(m)

Data: Module m
Result: A module is explored to generate its reachability graph

```
1  Tran ← ∅
2  foreach p∈P^m_port do  Tran ← Tran ⋃ T_out(p)
3  foreach t ∈Tran do
4  |  foreach p∈P_in(t) do
5  |  |  if PATTERN(E(p,t)) then PBB(t) ← PBB(t) ⋃ E(p,t)
6  |  end
7  |  C ← GetBindings(PBB(t))
8  |  foreach b∈C do
9  |  |  foreach p∈P^m_port ∧ p∈P_in(t) do  M_0(p) ← M_0(p) ⋃ E(p,t)⟨b⟩
10 |  end
11 end
12 i ←0
13 foreach p∈P_port do
14 |  ports[i] ← p
15 |  tokenCount[i] ← M_0(p).numOfTokens()
16 |  activeTokens[i++] ← 2^tokenCount[i]
17 end
18 while activeTokens[i-1]!=0 do
19 |  Process(M_0)
20 |  j ← 0
21 |  if activeTokens[j]!=0 then
22 |  |  activeTokens[j] = activeTokens[j] − 1
23 |  |  M_0(ports[j]).onlyTokens(activeTokens[j])
24 |  else
25 |  |  while activeTokens[j]!=0 & j<i do j++
26 |  |  if j≥i then break
27 |  |  activeTokens[j] = activeTokens[j] − 1
28 |  |  M_0(ports[j]).onlyTokens(activeTokens[j])
29 |  |  foreach k∈ [0 ··· j − 1] do
30 |  |  |  activeTokens[k] = 2^tokenCount[k]
31 |  |  |  M_0(ports[k]).onlyTokens(activeTokens[k])
32 |  |  end
33 |  end
34 end
```

Algorithm 10: Process(M_0)

Data: Initial state
Result: Generate reachability graph & access table

```
1  foreach p∈P^m_inport do accessTable.insertInPort(M0(p))
2  Unprocessed ← {(M0, "")}
3  Nodes ← {M0}
4  while !Unprocessed.empty() do
5      (M, conditions) ← Unprocessed.getNextMarking()
6      foreach ((t,b),M') such that M[(t,b)⟩M' do
7          if !Nodes.contains(M') then
8              Nodes.add(M')
9              Unprocessed.add((M',conditions))
10             if p∈P_outport∧p∈P_out(t) then
11                 accessTable.insertOutPort(M'(p),conditions)
12             end
13         end
14     end
15     b_param=conditions
16     foreach ((t,b)^ParaCond,?) such that M[(t,b)^ParaCond⟩? do
17         foreach E(p,t)∈ParaCond do
18             if E(p,t)⟨b_p⟩=true then b_param=Combine(b_param,b_p)
19         end
20         if G(t)∈ParaCond & G(t)⟨b_p⟩=true then b_param=Combine(b_param,b_p)
21         foreach E(t,p)∈ParaCond do
22             if E(t,p)⟨b_p⟩=true & M[(t,b_p)⟩M' then
23                 Unprocessed.add((M',Combine(b_param,b_p)))
24                 Node.add(M')
25                 if p∈P_outport then
                   accessTable.insertOutPort(M'(p),Combine(b_param,b_p))
26             end
27         end
28     end
29 end
```

Algorithm 11: onlyTokens(activeTokens)

Data: The value in activeTokens(*NOT* the reference)
Result: Select the tokens corresponding to this value

```
1 i←1;
2 while activeTokens!=0 do
3 |   if activeTokens>>1 == 1 then
4 |   |   include token numbered i in M0;
5 |   end
6 |   activeTokens←activeTokens / 2;
7 |   i++;
8 end
```

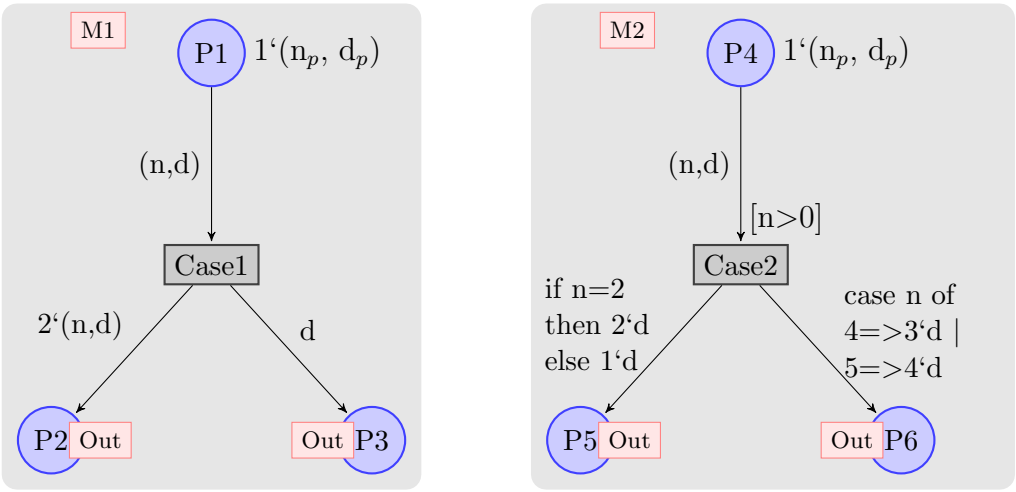


Figure 4.12: The two possible cases when generating a parametrised reachability graph.

In the first case, a binding b for a transition t is found that leads to a new marking M'. The set *Nodes* is scanned for M' and its absence ensures that the state was not explored previously. To process a new marking, it is added to *Unprocessed* along with the conditions that needs to be satisfied to reach that marking. Furthermore, if the new marking adds any tokens to an output port, it is reflected in access table.

The processing is similar in the second case, with the exception that the new marking M' cannot be determined unless the conditions are resolved. Consequently it is represented using a '*?*' symbol in step 16. Furthermore, *ParaCond* is assigned with the set of conditions associated with t that requires assigning specific values to at least one parameter. The subset of parameters that have already been assigned values are fetched from conditions in step 15. A condition in *ParaCond* could either be an input arc expression, output arc expression or a

Table 4.13: The entry in access-table for module M2

Conditions			**Output Ports**	
if n>0?	**if n=2?**	**case n of**	**P5**	**P6**
YES	YES	2=>	2‘d	empty
	NO	4=>	1‘d	3‘d
		5=>	1‘d	4‘d
NO		-	empty	empty

guard condition. For input arc expressions, specific values are assigned to parameters until they all evaluate to true (steps 17-19). If necessary, required values are assigned to additional parameters until the guard condition evaluates to true (step 20). The condition on output arcs is checked thereafter. Since the output arc expressions can contain one or more nested if/switch or case conditions, a different value of bindings leads to a different new marking M′. Each of these markings are stored into *Unprocessed* along with the condition (step 23). The latter is updated to contain the values that were assigned to the parameters. Furthermore, if any of the output places is a port, the contents are written into the access-table (step 25). The entry added to the access-table when both *P5* and *P6* in Figure 4.12 are output port places is shown in Table 4.13.

4.5.5 Additional memory cost for storing access-tables and parametrised reachability graphs

As observed earlier, each module has an access-table that is used by a substitution transition to determine the result of its execution. The rows in this table comprises of a combination of bindings that enable the output transitions of the input port places. Consider a module M with the output transitions of input port places in TS^M_{out}, where

$$TS^M_{out} = \{t_1, t_2, t_3, \cdots, t_{x-1}, t_x\} \tag{4.21}$$

For simplicity, we consider these transitions to be independent. This allows a transition in TS^M_{out} to be enabled for a binding irrespective of other transitions. Suppose that a transition t_i in TS^M_{out} is enabled by nb^{t_i} bindings. Therefore the number of possible combinations in which these bindings can enable t_i are

$${}^{nb^{t_i}}C_1 + {}^{nb^{t_i}}C_2 + \cdots + {}^{nb^{t_i}}C_{nb^{t_i}} \tag{4.22}$$

Equation 4.22 follows from the fact that any one or more of these bindings can enable the transition. However, the only way of not enabling the transition would be to use none of these bindings, as shown in equation 4.23.

$$^{nb^{t_i}}C_0 \tag{4.23}$$

The total number of combinations in which a transition can either be enabled or disabled is the sum of equations 4.22 and 4.23.

$$=^{nb^{t_i}}C_0 +^{nb^{t_i}} C_1 +^{nb^{t_i}} C_2 + \cdots +^{nb^{t_i}} C_{nb^{t_i}} \tag{4.24}$$

$$=2^{nb^{t_i}} \tag{4.25}$$

Enabling a module requires enabling any one of the transitions in TS^M_{out}. This in turn requires at least one transition in TS^M_{out} to use an enabling binding. Therefore the total combination of bindings that would enable a module would be all possible combinations except the one that disables all transitions in TS^M_{out}.

$$TOT_M = 2^{nb^{t_1}} * 2^{nb^{t_2}} * \cdots 2^{nb^{t_x}} - 1 \tag{4.26}$$

$$= 2^{nb^{t_1}+nb^{t_2}+\cdots+nb^{t_x}} - 1 \tag{4.27}$$

This is nothing but the number of entries in access-table. If each entry occupies α space, the total memory needed to store the access table would be $TOT_M * \alpha$.

Figure 4.13 shows the increase in space requirement with an increase in the number of enabled bindings. Considering the sharp rise in curve, our technique might not be suitable for modules that have large number of output transitions for input port places.

Each row of access-table is associated with a parametrised reachability graph that also requires additional memory. When the substitution transition uses a particular row of an access-table, the corresponding parametrised reachability graph is also fetched. This is done in order to account for the states that could be reached when actually executing the module.

When a substitution transition fetches a parametrised reachability graph, it assigns the parameters to determine the new states. These states are then stored in the memory for detecting duplicate states (as explained in previous chapter). If a particular row of the access-table is used only once by the substitution transition, each state in parametrised reachability graph has a counterpart in memory (after assigning values). Therefore the memory required

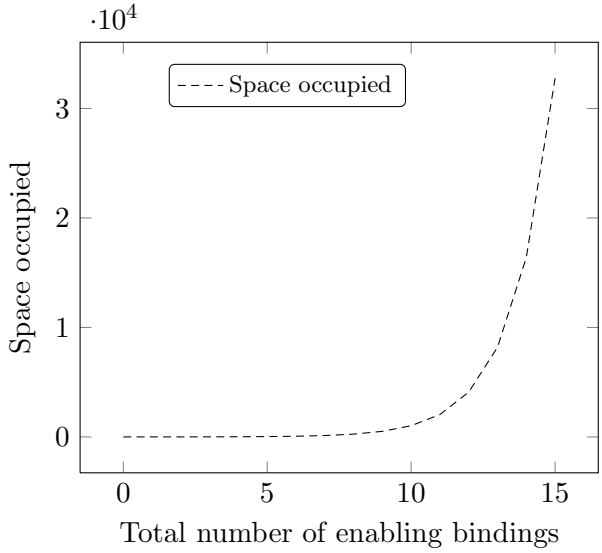


Figure 4.13: The space occupied by access table increases with the number of enabling bindings.

is twice what it would be otherwise. However, if the row is accessed twice, each state in parametrised reachability graph would have two other counterparts in memory. Therefore an addition 1.5 times of memory is required in this case. In general, the percentage of additional memory required would decrease with an increase in average use of parametrised reachability graphs. This is shown in Figure 4.14

4.5.6 Theoretical evaluation of the reduction in delay

The technique proposed herein reduces the delay in model-checking by exploring modules in parallel. However, the degree of reduction depends on a range of factors. For instance if 1)the root module does not have any enabled transitions and 2) the tokens in its socket places do not enable any sub-module, the delay in model-checking might increase. This is essentially because the access-tables and parametrised reachability graph for each sub-module would anyway be created. This section evaluates the reduction in delay offered by the proposed technique.

Consider a row i in the access-table for module M whose parametrised reachability graph was derived in time x. When a substitution transition uses this row, it also fetches the associated parametrised reachability graph to assign the parameters and determine new states. Considering that this does not require executing any transitions, the time required should be

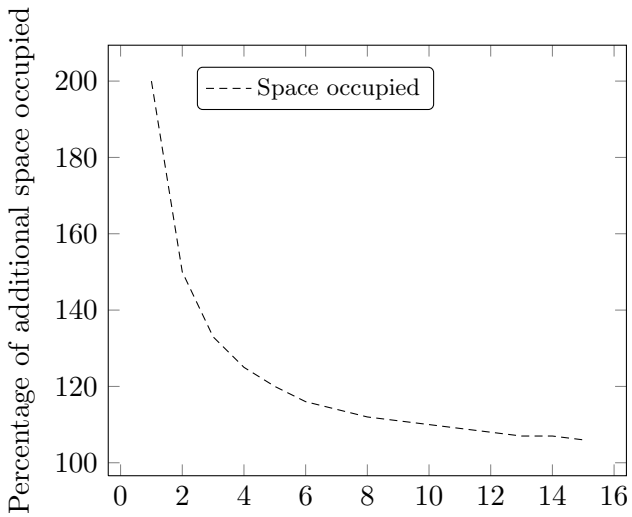


Figure 4.14: The percentage of additional space occupied decreases with an increase in usage of parametrised reachability graphs.

a fraction of x ($=\beta * x : 0 < \beta < 1$). Without using our technique, the substitution transition could have executed the transitions in the sub-module and determined the new states in time x. Therefore the difference in time delay is

$$x - (x + \beta * x + \alpha) = -(\beta * x + \alpha) \tag{4.28}$$

where α is the time to access a row in the access-table. The negative result indicates that the delay has increased when using our technique. However, if the row i is accessed twice, the difference in time delay is

$$2x - (x + 2 * \beta * x + 2 * \alpha) = x - 2 * (\beta * x + \alpha) \tag{4.29}$$

Considering that β and α are substantially smaller than x, equation 4.29 should yield a positive difference in delay. The value of β that ensures a positive difference in delay is derived using equations 4.30 - 4.33. From equation 4.29, a positive difference in delay requires

$$x - 2 * (\beta * x + \alpha) > 0 \tag{4.30}$$

$$or \quad x * (2 * \beta - 1) < -2 * \alpha \tag{4.31}$$

Since β and α are positive constants and x is a positive variable, equation 4.31 holds iff

$$2 * \beta - 1 < 0 \tag{4.32}$$

$$or \quad \beta < \frac{1}{2} \tag{4.33}$$

In general, as shown in equations 4.34 and 4.35, the difference in delay follows an identical pattern as more substitution transitions use row i.

$$3x - (x + 3 * \beta * x + 3 * \alpha) = 2x - 3 * (\beta * x + \alpha) \tag{4.34}$$

$$cx - (x + c * \beta * x + c * \alpha) = (c - 1)x - c * (\beta * x + \alpha) \tag{4.35}$$

The general equation 4.35 returns a positive difference in delay when

$$x(c - 1) - c * (\beta * x + \alpha) > 0 \tag{4.36}$$

$$or \quad x(c * \beta - c + 1) < -c * \alpha \tag{4.37}$$

Since β, α and c are positive constants and x is a positive variable, equation 4.37 holds iff

$$c * \beta - c + 1 < 0 \tag{4.38}$$

$$or \quad \beta < \frac{c - 1}{c} : c > 0 \tag{4.39}$$

Equation 4.39 justifies the negative difference of delay in equation 4.28 when c=1. Furthermore, equation 4.42 renders the value of β for which equation 4.35 is strictly increasing.

$$\frac{d}{dx}((c - 1) * x - c * \beta * x - c * \alpha) > 0 \tag{4.40}$$

$$or \quad c - 1 - c * \beta > 0 \tag{4.41}$$

$$or \quad \beta < \frac{c - 1}{c} : c > 0 \tag{4.42}$$

The conditions for positive difference in delay and its strict increase in value are found to be the same.

In order to determine the *O()* function, we consider a model with m modules which are used M times (in some combination) during the state space analysis. If the average probability of using module K_i during state space analysis is *p(i)*, it is used $M * p(i)$ times

in total. Using equation 4.35, the total time taken by module K_i can be calculated as

$$T_i = x_i + M * p(i) * \beta_i * x_i + M * p(i) * \alpha_i \tag{4.43}$$

where the symbols have their usual meaning and the subscripts denote the module number. Consequently the total time taken for state space exploration is

$$T = T_1 + T_2 + \cdots + T_m \tag{4.44}$$

$$= \sum_{i=1}^{m} x_i + M * \sum_{i=1}^{m} p(i) * \beta_i * x_i + M \sum_{i=1}^{m} p(i) * \alpha_i \tag{4.45}$$

Since in most cases the modules of a model would be heavily used,

$$M * \sum_{i=1}^{m} p(i) * \beta_i * x_i >>> \sum_{i=1}^{m} x_i \tag{4.46}$$

where $>>>$ denote *far exceeds*. Furthermore, since the time for executing a module (x_i) far exceeds the time to read the access table (α_i),

$$M * \sum_{i=1}^{m} p(i) * \beta_i * x_i >>> M \sum_{i=1}^{m} p(i) * \alpha_i \tag{4.47}$$

Therefore the expression in equation 4.45 that determines the delay is

$$M * \sum_{i=1}^{m} p(i) * \beta_i * x_i \tag{4.48}$$

Considering that M and β are constant for a model, the total time essentially depends on the product of probability and the time of execution for each module. Consequently

$$T = O(px) \tag{4.49}$$

4.6 Results

To evaluate the proposed technique for time-efficient state-space analysis, we have implemented a model-checker from scratch that incorporates the proposed technique. Instead of a full-fledged model-checker, we have implemented a cut-down version that can handle only integer colour-sets. Furthermore, the model-checker assumes the value in each token to be

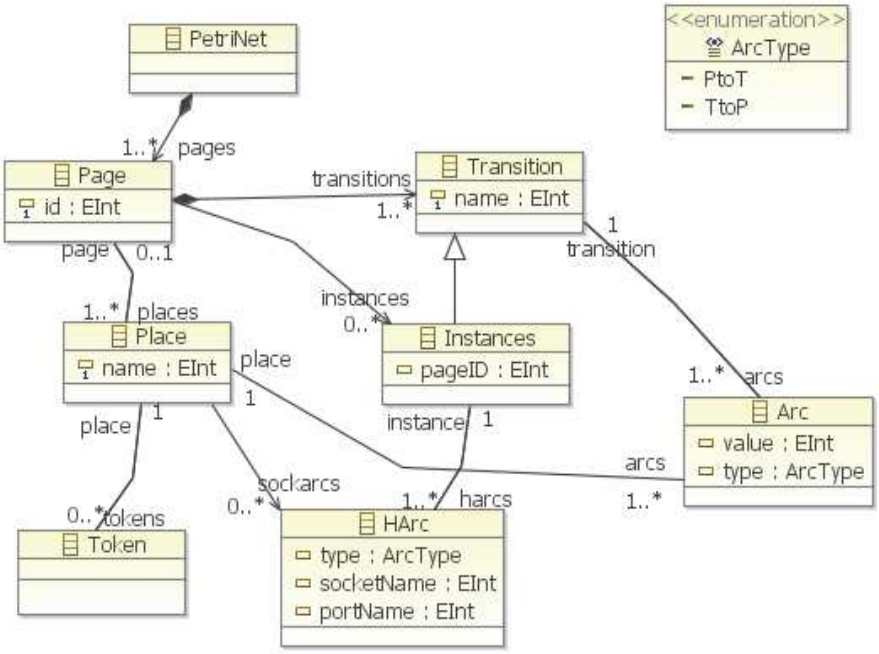


Figure 4.15: The object model implementation using EMF.

one. Such a set-up simplifies the experiment without capitulating any advantages offered by a model-checker. The results obtained from CPN tools [Jensen et al., 2007] have been used as a benchmark for comparison. In order to ensure a fair comparison, both the model-checkers explore the same HCPN model shown in Figure 4.16.

4.6.1 Experimental Setup

The model-checker was implemented using Java (Standard Edition, Runtime Environment Version 6) programming language. The object model was created using Eclipse Modeling Framework (EMF) [emf, 2010] which generates java classes for the model, in addition to representing it structurally. It also generates a set of adapter classes that enable viewing and command-based editing of the model.

Figure 4.15 illustrates the object model that contains the classes corresponding to the various elements in a CPN model. These classes need to be instantiated and their properties initialised before the CPN model can be model-checked. In this pursuit, we created an importer that accepts a CPN model as an XML document and parses it to instantiate these classes. However, it is also possible to instantiate these classes programmatically and assign

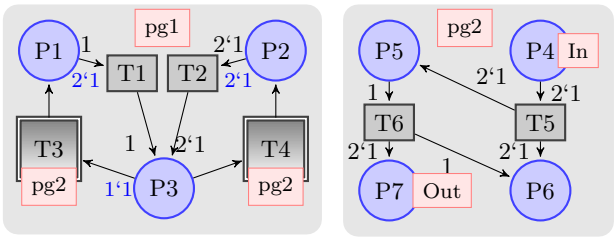


Figure 4.16: HCPN model used for evaluation.

their properties. All attempts have been made to ensure that our object model is similar to that of Access/CPN [Westergaard and Kristensen, 2008]. Such a similarity ensures the portability of a CPN model from our implementation into CPN tools [Jensen et al., 2007].

The class corresponding to a Petri-Net is known as *PetriNet* and sits at the top of the hierarchy. It can have one-or-more *Pages*, each corresponding to a module. A *Page*, in turn, has one-or-more *Places* and *Transitions*. Additionally, it can also have any number of *Instances*, the CPN tool's equivalent of substitution transition and a subclass of *Transition*.

Each arc has a corresponding class that stores its arc-expression and arc-type. Since we consider only integer tokens, the arc-expression is always an integer. An arc-type could either be *PtoT* or *TtoP*, defined by enumeration *ArcType*.

The class *HArc* corresponds to arcs connecting a place and an instance. In addition to arc-type, it also has the name of its adjoining socket-place and the port-place associated with it. The arc-value is not stored as it holds no significance in this context.

Furthermore, the class *Token* corresponds to a token in a CPN model. For simplicity, the value of each token is assumed 1.

4.6.2 Empirical Results

The HCPN model shown in Figure 4.16 is explored by the aforesaid model-checkers to produce the results. As required by the simplifications in our implementation, all the places in this model have integer colour-set and all tokens in them have value '1'.

Figure 4.17 compares the time taken in model-checking the HCPN model, both with and without using our time-efficienct state-space analysis technique. The latter is obtained using CPN tools, currently the default model-checking tool for HCPN models. While x-axis records the number of markings generated(in thousands), the time taken to generate them is accounted along y-axis. Considering that the model has infinite state-space, the plots are restricted to the time taken in generating the first 25,000 unique markings.

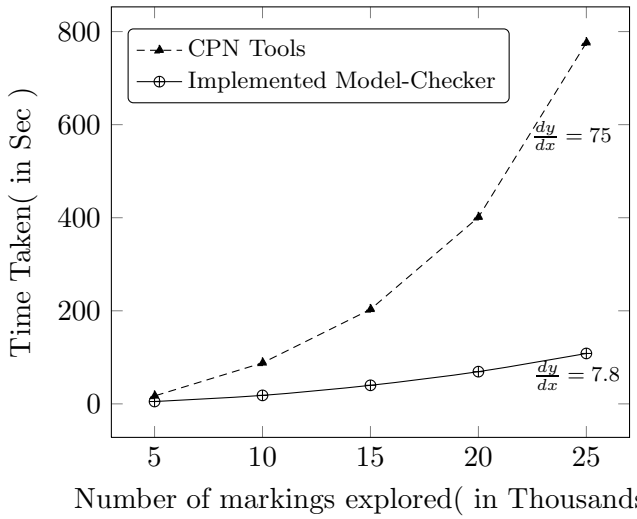


Figure 4.17: Time taken for model-checking the HCPN model shown in Figure 4.16.

Figure 4.17 clearly exhibits the time reduction offered by our time-efficient technique. As compared to CPN tools, the implemented model-checker offers a massive 86% reduction for generating the first 25,000 markings. Considering that the slope for its plot(=7.8) is $\frac{1}{10}$ that for CPN tools(=75), the offered reduction would only increase as more markings are generated.

The proposed technique require some additional memory to store access-tables and parametrised reachability graphs. This memory is insignificant when compared to the memory occupied by state-space of a HCPN model. Consequently, the cost of this additional memory is not accounted.

4.7 Discussion

The experimental results indicate a massive reduction in time requirements when using the implemented model-checker. Considering that the proposed algorithms do not make any assumptions about the value in tokens or the data-type(or colour) of places, a full-fledged model-checker incorporating our algorithm is also expected to deliver identical reduction.

The example HCPN in Figure 4.16 has two modules, *pg1* and *pg2*. When using the proposed algorithms, an access-table and a parametrised reachability graph is created for *pg2*. No such tables or graphs are created for *pg1* because the model does not have a module that is dependent on it. The substitution transitions *T3* and *T4* execute by probing the

Table 4.14: The access-table for pg2

Input Ports	Conditions	Output Ports
P4	**-**	**P7**
2‘1	-	4‘1

access-table for *pg2* shown in Table 4.14. This in turn prevents *T5* and *T6* from firing and accounts for the reduction in time requirement.

As discussed previously, there is an additional memory requirement to store the access-table and parametrised reachability graphs. However, this being a very small model, the additional space requirement was negligibly small and could not be measured and taken into account. Furthermore, the additional space need not necessarily be measured even for larger models. This is because the memory occupied by the access-table and parametrised reachability graph is static and does not change as the state-space is explored.

As observed previously, state-space exploration involves determining the reachable states of the system and investigating them for a set of undesirable properties. To ensure efficiency, the model-checker maintains a list of explored states and rejects any duplicate state. The x-axis in Figure 4.17 only accounts for unique states of the system that were reached by the model checker. However, its y-axis accounts for the time in producing both the unique and duplicate markings. This is because a model-checker needs to generate a marking and compare it with the stored states before determining if it is new. Regardless of the technique used for this comparison, the model-checker spends time in processing duplicate states. Therefore a surge in duplicate markings would lead to a rapid increase in y-coordinate, propelling a steep increase in the slope of the associated curve. For large models, the proposed algorithms offers massive reduction owing to its ability to reduce the time requirement for producing both duplicate and unique markings alike.

4.8 Summary

In this chapter, we have proposed a technique to reduce the time requirement for model-checking a hierarchical model. The reduction obtained is attributed to generating the reachability graph for each module in parallel. Experimental results indicate a time-reduction of 86% as compared to CPN tools when generating the first 25,000 states. The technique is found to have a very small time overhead, negligible for practical situations, at the expense of a small increase in memory requirement. Considering the inherent complexities of modern

software systems, more and more model-checkers ought to embrace a hierarchical representation in order to ensure multiple levels of abstraction. Our algorithm is addressing a niche for such systems.

Chapter 5

Generating Hierarchical Models by Identifying Structural-Similarity

Model-Checking requires formulating a formal representation of the system prior to verifying it. Considering the parallel components in a contemporary service composition, this translation often leads to awful increase in the size of the obtained representation that eventually becomes a computational bottleneck in model-checking algorithms. Such a massive model is difficult to draw and impractical to analyse and maintain. Consequently, it is prone to errors and omissions that impair the benefits of model-checking. Furthermore the lack of abstraction and classification in such voluminous formal models oblige a human modeler to circumvent a thorough understanding.

In order to obtain a more succinct representation with multiple levels of abstraction, the system needs to embrace the notion of hierarchy [Alur and Yannakakis, 2001]. In a hierarchical setup, each system component is represented by a module wherein the module for a high-level component refers to its underlying components using their module name or reference. Apart from rendering an elegant, abstract and expressive model, such a setup also allows alleviating the state-space explosion by applying *compositional model-checking* [Clarke et al., 1989].

In this chapter, we propose a *decrease-and-conquer* based method for installing hierarchy into an otherwise 'flat' model. The method involves determining the structurally similar components in a flat-model and creating a module for each one of them. A decrease-and-conquer based strategy breaks the bigger problem into a set of smaller problems and the solutions to smaller problems are combined to solve the original problem. Apart from the

aforementioned benefits, such a method also helps in extending the time-efficient state-space analysis technique proposed in Chapter 4 for non-hierarchical models. The experimental results indicate a linear time complexity, namely *O(n)*, where n is the number of nodes in the flat-model. Furthermore, as opposed to the related techniques [Berthelot, 1986; Haddad, 1990; Evangelista et al., 2005], our technique ensures that the transformed model is equivalent to the original model.

5.1 Motivation

Model-checking is an automatic verification technique that is being rapidly embraced for quality assurance of software-systems. However, a model-checker requires a formal representation of the system in order to verify it. Considering that the components in a *component-based system* [Leavens and Sitaraman, 2000] could be arbitrarily nested, such a representation would be enormously large for human comprehension. A large model[1] would be difficult to draw and impractical to analyse and maintain [Jensen and Kristensen, 2009]. The crux of the problem is the set of nested components that require the representation of a low-level component to be added once for each overlaying component using it.

Alternatively, a model could be constituted out of a set of modules wherein each module represents a system component. In such a hierarchical setup, the module for a high-level component refers to its underlying components using their module name or reference. This avoids the acute increase in model size owing to the inclusion underlying components in actual representation. Furthermore, the benefits increase with each additional high-level component sharing an underlying component. Consequently the obtained model would be significantly more succinct owing to the notion of hierarchy introduced. In addition, modifying a component would only require altering the corresponding module.

Recently there has been a number of attempts in auto-generating the formal representation of software systems [Chen and Cui, 2004; Fu et al., 2004]. The primary objective in auto-generating a model is to produce the input for a model-checker. Regardless of the structure of input (flat or hierarchical), the model-checker verifies the system under deliberation. Consequently these tools often render flat models as there is limited incentive in introducing hierarchy. However, the rendered model might assist in accomplishing additional objectives like identifying the overall architecture of the system, understanding its dependencies, visualising the flow of information through it, identifying its capabilities and limitations

[1] model is used synonymously to a formal representation

and calculating its complexity [Christopher, 2003]. Nevertheless, it might be impossible to scrutinise the flat-model because of its massive size. This chapter proposes new methods for accomplishing these additional objectives by introducing hierarchy into a flat-model and rendering it exponentially more succinct.

Some solutions based on *pre-agglomeration* [Haddad, 1990] and *post-agglomeration* [Haddad, 1990; Evangelista et al., 2005] reduce the size of a model by merging some sequential events. However, these reduction techniques primarily address the problem of state-space explosion [Christensen et al., 2001] by reducing the number of execution traces to be analysed. Consequently the technique itself, as well as the reduction it offers depend on the property to be analysed when exploring the state-space. Furthermore, the reduced model obtained using these solutions differ from the original model. In Chapters 3 and 4, we have proposed techniques to alleviate the state-space explosion problem. Subsequently the solution proposed herein installs hierarchy into a given model to obtain an equivalent concise model.

The proposed solution has two distinct parts 1) a *Lookup method* that identifies the set of structurally similar components in a model and 2) a *Clustering method* that establishes hierarchy thereupon. The Lookup method is based on the 'Decrease-and-Conquer' [Puntambekar, 2008] strategy wherein the bigger problem is broken into smaller problems and the solutions to smaller problems are combined to solve the original problem. Consequently it starts by identifying the smallest components in a model that are identical. Thereafter it progressively determines the larger components by recursively attaching the adjoining elements of the identical components determined in the previous step and comparing them for similarity. This in-effect translates to identifying the fine-grained components in the system followed by the determination of their overlaying components bottom-up. Later these components are mapped into modules by the Clustering method in order to establish a hierarchy.

A decrease-and-conquer algorithm requires the solution of at least one sub-problem in order to use it and determine the solution of larger problems. This is different from the divide-and-conquer technique wherein the solution of several sub-problems are required. Considering the ease in determining and solving the smallest sub-problem of the original problem, decrease-and-conquer has been selected as the appropriate design technique. The technique for solving the smallest problem is included in the Lookup method.

Our contributions can be summarised as:

1. A *Lookup method* that identifies the structurally similar components of a model. Essentially it identifies the identical components in a model that would be used for creating individual modules of an equivalent hierarchical model. The algorithm for this method has a linear time complexity for sufficiently large CPN models (up to 141 places and 132 transitions).

2. A *Clustering method* that establishes hierarchy over a flat model by forking a module out of each identical component identified by the Lookup method. We discuss it only in the context of establishing hierarchy in Coloured Petri-net models.

Although a wide array of languages are available to model a software-system, each model-checking tool essentially supports only a specific modeling language. Some common modeling languages along with the supported model-checking tool are *Promela* for *SPIN* [spi, 2007], the *C* programming language for *BLAST* [Beyer et al., 2007] and *Coloured Petri-nets* (CPN) for *CPN Tools* [Jensen et al., 2007]. Considering the subtle differences between these modeling languages, it is difficult to propose a generic method for installing hierarchy. Consequently the Lookup and the Clustering methods proposed in this chapter specifically target CPN models. Our methods offer no additional advantage in using the CPN models and they can be independently adapted for any other language that define a notion of hierarchy and structural similarity.

Throughout the chapter, a CPN or a Petri-Net model is sometimes referred as a net while their sub-parts are referred as subnets or components.

The rest of the chapter is organised as follows. Section 5.2 introduces the deliberated problem and provides an insight into the presented solution. Thereafter Section 5.3 introduces the basics of substitution transitions. Prior to proposing the Lookup and the Clustering methods in Section 5.5, the related work is compiled in Section 5.4. We plot the experimental results in section 5.6 and discuss the outcome in section 5.7. Finally we summarize our contributions in section section 5.8.

5.2 An Overview of the Deliberated Problem & the Proposed Solution

This section presents the problem briefly and outlines the proposed solution. As pointed out previously, a system needs to be modeled using one of the several available modeling languages prior to generating and analysing its state-space. This being a tedious and error-prone

activity, several techniques have been proposed to auto-generate the formal representation of software systems [Chen and Cui, 2004; Fu et al., 2004]. However, as mentioned previously, the complex and concurrent components in contemporary systems leads to abysmal increase in model size. Many of the modeling languages support hierarchical constructs and offer multiple levels of abstraction. However, there are limited incentives in using a hierarchical modeling language and enhancing the human understandability and modularity of the rendered model. The primary objective in formalising a system is to produce the input for a model-checking tool that is indifferent to the orientation and structure of the formal model. Nevertheless, we identify the advantages of introducing hierarchy and modularity in a formal model. A formal model might assist a human modeler in identifying the overall architecture of the system, understanding its dependencies, visualising the flow of information through it, identifying its capabilities and limitations and calculating its complexity [Christopher, 2003]. However, the flat model produced by auto-generating techniques pose a serious challenge in accomplishing these objectives. This chapter proposes a novel method for realizing these objectives by introducing hierarchy into a flat-model and rendering it more succinct.

A software system is usually composed of a set of components and on formalizing it using an auto-generating tool, the rendered model has footmarks of the individual components [Wohed et al., 2002]. Consequently any effort in identifying the structurally similar sub-parts of the model should yield the models corresponding to individual components. These models can thereupon be used to create the modules of a hierarchical model. This follows from an earlier discussion in Section 5.1 wherein hierarchical models were affirmed to consist of modules corresponding to the components of a system.

Some of the steps of the proposed technique are illustrated in Figure 5.1. Initially the flat model is scrutinised to determine the set of identical sub-parts. These sub-parts corresponds to the components of the overlying system. Nevertheless, modeling this system directly using a hierarchical formalism will render a representation consisting of modules corresponding to each of these components. Consequently the flat model could be transformed into its equivalent hierarchical model by creating modules from these identified sub-parts. As shown in Figure 5.1, this is done by replacing these sub-parts in the model with a stub to the actual module. The aforementioned transformation involves two discrete steps: 1) to identify the structurally similar sub-parts of the model, and 2) to use the sub-parts in creating the modules of a hierarchical model. While the first step is accomplished by using the Lookup method, the Clustering method is used subsequently.

The Lookup method is based on the decrease-and-conquer paradigm and it determines

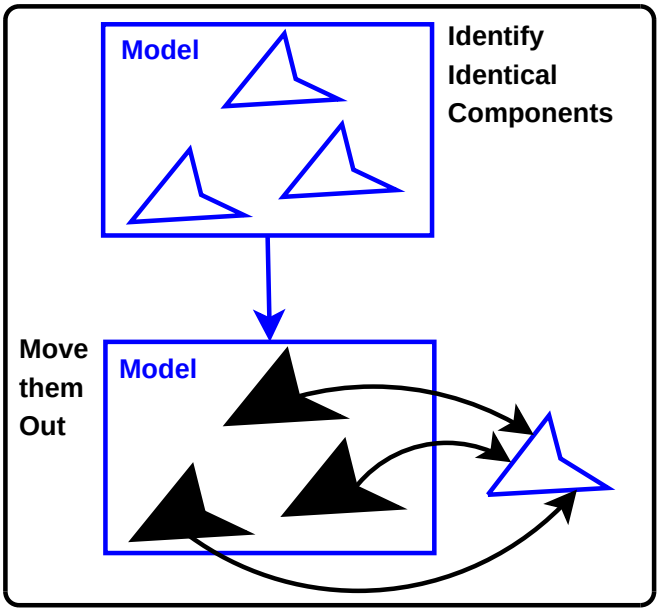


Figure 5.1: The identical components are identified and moved out.

the identical components bottom-up starting from the elementary sub-parts of the model. It is worth mentioning that these elementary sub-parts depend on the modeling language being used. In case of graph (or bipartite graph e.g. CPN) based models, these are essentially the vertices of the graph. Consequently the smallest problem boils down to finding the identical vertices in a graph as defined in Definition 14. The smallest problem is solved by classifying each disjoint set of vertices based on their indegree and outdegree. This yields the set of smallest identical components that consists of single vertices. Thereafter the solutions to the smaller problems are used in solving the bigger problem. In this context, the bigger problem is to determine even larger components. In each step henceforth, the adjoining nodes of these similar vertices are attached before comparing the rendered components. Considering that the existing components were already compared in a previous step, the Lookup method can determine the similarity of new components by comparing the last attached nodes.

Finally the Clustering method generates a module for each identified component to generate the corresponding hierarchical model. This again depends on the semantics of hierarchical modeling language being used. Since we use CPN in this chapter, the hierarchy is established using the semantics of *Substitution Transition* for Coloured Petri-nets.

As compared to other transformation techniques [Berthelot, 1986; Haddad, 1990; Evangelista et al., 2005], the Lookup and the Clustering methods render an equivalent hierarchical

model. In addition, they address a niche for contemporary software systems that constitutes of parallel components. Identifying and modeling individual components helps the human modeler in understanding the system. Furthermore, the transformation renders a valid hierarchical CPN model that could be both simulated and verified using CPN tools.

5.3 Basics of Substitution Transition

This section briefly explains the semantics of Substitution transition for CPN that are used by the Clustering method. As discussed previously, a hierarchical model consists of a set of modules. However, unless these modules are correlated, they cannot constitute a formal model. The substitution transitions act as stubs to associate and connect these modules.

Each module of a hierarchical CPN model has a substitution transition as its proxy. A module using the services of another module has the proxy for the latter as a substitution transition. On executing this transition, the corresponding module executes and delivers the required services. Consequently replacing each proxy with its underlying module do not alter the behaviour of the formal model. However, this would destroy the hierarchy and render a flat model.

This is further explained using Figures 5.2 and 5.3. Figure 5.2 shows a CPN model with two identical components, (A-T1-F-T3-G-T2-B) and (C-T4-H-T6-I-T5-D). These components are replaced using substitution transitions *Hier1_1* and *Hier1_2* in Figure 5.3. Furthermore, the components themselves have been moved out to constitute a separate module. The module that contains a substitution transition is called a *Supermodule* while the substituted module is called a *Submodule*. In Figure 5.3, *Page1* is a supermodule while *Subpage1* is a submodule. In addition, each proxy has its own instance of the submodule. For example the two instances of submodule *Subpage1* (i.e. *Subpage1(1) and Subpage1(2)*) in Figure 5.3 corresponds to the substitution transitions *Hier1_1* and *Hier1_2*.

A supermodule and its submodules are glued by defining a one-to-one relationship between a subset of their places. Each place adjacent to a substitution transition is known as *socket* and has a counterpart in the associated submodule that is known as *port*. Furthermore, considering that a socket could either be an input, output or I/O place of the substitution transition, an equivalent tag is attached to the corresponding port to indicate its permitted type. A port assigned *In* can only be associated with an input place of the substitution transition. Similarly a port assigned *Out* can only be paired with an output place of the substitution transition. A port assigned *I/O* can be associated to a socket which

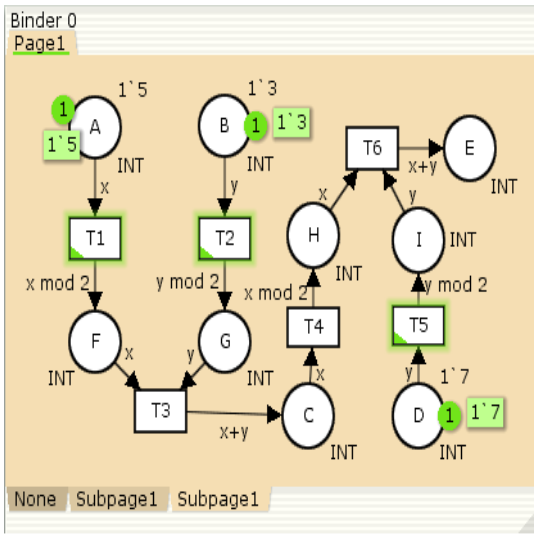


Figure 5.2: A Coloured Petri-Net model with identical components.

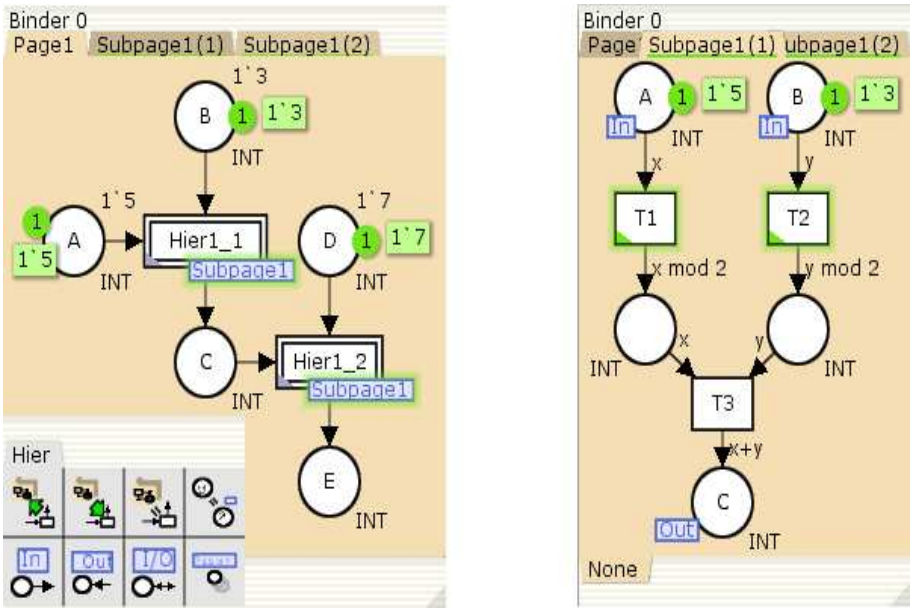


Figure 5.3: The CPN model in Figure 5.2 with hierarchy installed using substitution transition. Page1 is the supermodule while Subpage1(1) and Subpage1(2) are the two instances of submodule corresponding to the two substitution transitions.

is both the input and output place of the substitution transition. For instance, port A in Figure 5.3 is assigned port-type *In* and consequently linked to socket A, which is an input place of the substituting transition.

The services of a submodule can be used by any number of supermodules by including them as a substitution transitions. This allows reusing a defined submodule and forms the basis of the reduction method proposed in this chapter. However, as mentioned earlier, a separate instance of the submodule is created for each substitution transition.

5.4 Related Work

A set of transformations proposed in [Berthelot, 1986] aim to reduce the size of a Petri-Net model by merging two or more of its places or transitions based on certain conditions. These transformations preserve several classical properties of nets (like boundedness, safety, liveness etc) and also reduce the number of reachable states when performing state-space analysis. The implicit place simplification and pre & post agglomeration of transitions are the most frequently used transformations in [Berthelot, 1986] and were extended for coloured Petri-Nets in [Haddad, 1990]. In implicit place simplification, a place is removed if its marking is always sufficient to allow firing of any transition attached to it. In pre-agglomeration, two transitions t_1 and t_2 are merged if a place p is the sole input place for t_2 and is also an output place for t_1. Likewise in post-agglomeration, two transitions t_1 and t_2 are merged if a place p is the sole output place of t_1 and is also an input place for t_2. More recently, [Evangelista et al., 2005] proposed coloured Petri-Net reductions based-on post-agglomerations.

Table 5.1 compares the features of these related algorithms with the proposed method. Apart from reducing the size of the model, the transformations in [Berthelot, 1986; Haddad, 1990; Evangelista et al., 2005] also diminish the state-space by reducing the number of execution traces to be analysed. However, the net obtained after transformation is neither equivalent to the original net nor preserves properties of original net other than those specifically targeted. Consequently, we address the problem in two separate steps. The Lookup and the Clustering methods proposed in this chapter reduce the size by transforming a net into and equivalent hierarchical model.

5.5 The Proposed Technique for Installing Hierarchy

This section describes a method for identifying the structurally similar components in a CPN model and installing hierarchy into it. Considering that a CPN model is a folding [Jensen,

Table 5.1: A comparison of related reduction methods

Method	Resultant net same as original net	Model size reduced	State-Space reduced	Properties preserved
[Berthelot, 1986]	⊗	●	●	◐
[Haddad, 1990]	⊗	●	●	◐
[Evangelista et al., 2005]	⊗	●	●	◐
[Mukherjee et al., 2010]	●	⊗	●	●
The Lookup Method	●	●	⊗	●

●→Satisfies ◐→Partially satisfies ⊗ →Does not satisfy

1996] of an equivalent Petri-Net (PN), the technique is immediately applicable for the latter. Although the proposed technique is discussed only in the context of CPN language, it could be extended for other modelling languages that have the notion and semantics of hierarchy and structural similarity.

The technique proposed herein is applicable for modern software systems that embrace a component based model as opposed to a monolithic model. While the former allows separating discrete functionalities and features of a system as separate components that could be reused, the latter focuses on building an indivisible monolithic system that is difficult to use and maintain, despite being functionally equivalent to their component based counterparts. After auto-generating a model, the Lookup method helps in determining the structurally similar subparts in it that would correspond to the components in a system. The hierarchy can be installed thereafter by formulating a module out of these individual identical components.

5.5.1 The Lookup Method

This section proposes the Lookup method to determine the identical components in a model. In order to ensure better understanding, it is also enacted on an example net shown in Figure 5.6. Furthermore, the steps of the Lookup method are listed as an agglomeration of three related short algorithms instead of a single lengthy algorithm. This allows the reader to deal with fewer details simultaneously. As illustrated in Figures 5.4 and 5.5, the three algorithms constitute the two phases of the Lookup method. These phases are briefly introduced herein before analysing them in subsequent sections.

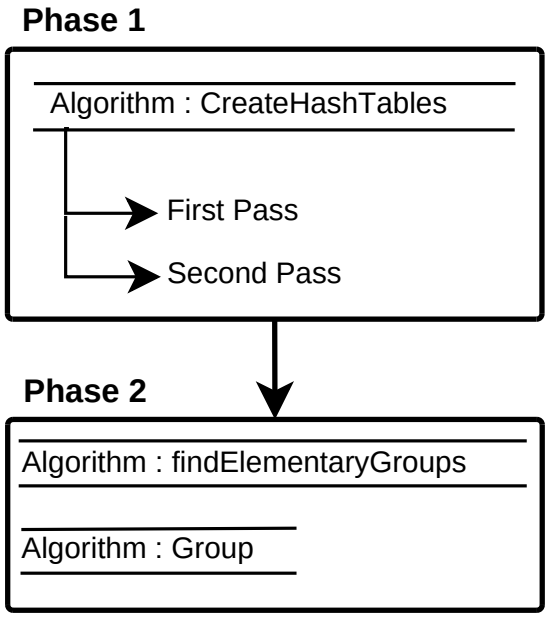


Figure 5.4: The three algorithms constituting the two phases of the Lookup method.

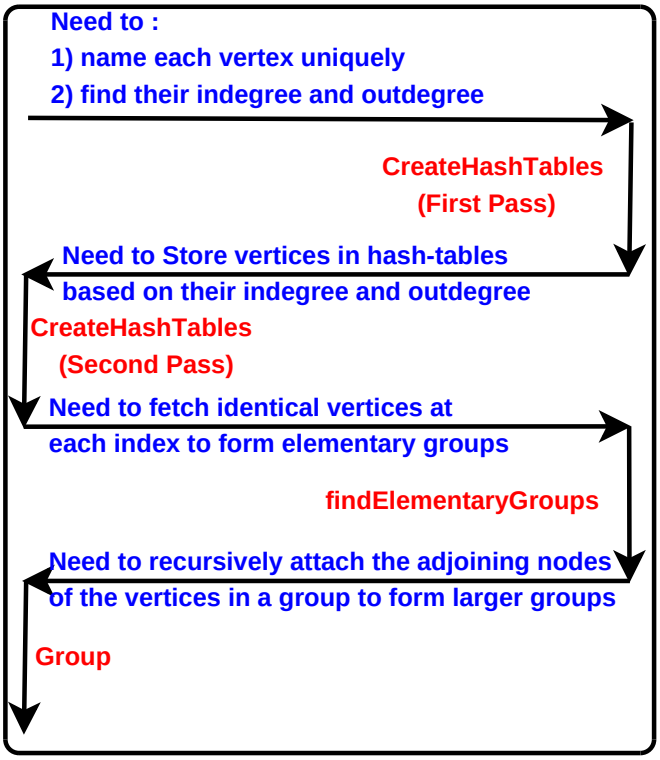


Figure 5.5: The roadmap of the proposed solution. Blue indicates requirement that is addressed by the immediately following Algorithm (in red).

As stated previously, the Lookup method is based on the 'Decrease-and-Conquer' [Puntambekar, 2008] strategy wherein the bigger problem is broken into smaller problems and the solution to smaller problems are combined to solve the original problem. The first-phase, in effect, solves the smallest problem of finding the structurally similar vertices in a graph-based model.

Definition 14 *A set of vertices* $\{v_1, v_2, \cdots, v_k\}$ *in a directed graph* $G = (V_1, V_2, \cdots, V_n, E)$*, with* n *disjoint sets of vertices* $\{V_1, V_2, \cdots, V_n\}$*, are said to be identical iff 1)* $\forall i \in (1,k)$, $\exists j \in (1,n) : v_i \in V_j$*, and 2)* $\exists i \in (1,k)$, $\forall j \in (1,k)$ *: Indegree*(v_i) *= Indegree*(v_j) *and Outdegree*(v_i) *= Outdegree*(v_j)*.*

Corollary 6 *If the edges* $\{e_1, e_2, \cdots, e_m\} \subseteq E$ *lead to vertex* $v \in V$ *in a graph* $G = (V,E)$*, then Indegree*$(v) = |\{e_1, e_2, \cdots, e_m\}|$*. This corollary is used in Algorithm 12 to determine the indegree of vertices.*

Corollary 7 *If the edges* $\{e_1, e_2, \cdots, e_m\} \subseteq E$ *lead away from vertex* $v \in V$ *in a graph* $G = (V,E)$*, then Outdegree*$(v) = |\{e_1, e_2, \cdots, e_m\}|$*. This corollary is used in Algorithm 12 to determine the outdegree of vertices.*

From Definition 14, the smallest problem can be solved by classifying the vertices in each disjoint set based on their indegrees and outdegrees. Nevertheless, the indegree and outdegree for each vertex need to be determined before such a classification. As shown in Figure 5.5, the first pass of Algorithm 12 scrutinises the vertices to determine these values.

However, classifying these vertices solve only half of the problem. These vertices must thereupon be stored in an appropriate data structure to allow constant time lookup. This would allow the next phase to efficiently fetch the results from this phase and solve the bigger problem. Considering that hash-tables have constant time lookup, they are used to store the vertices wherein each possible combination of indegrees and outdegrees map into a distinct index. Furthermore, for simplicity, each disjoint set of vertices are stored into a separate hash-table. As shown in Figure 5.5, the second pass of Algorithm 12 populates the hash-tables. It should be noted that the hash-tables store a list of vertices at each index and the hash function ensures that these vertices are all structurally similar.

The identical vertices determined in first phase form the smallest components. The second phase thereupon steps in and recursively attaches the adjoining nodes of these vertices to create larger components. As shown in Figure 5.5, Algorithm 13 fetches the identical vertices from each index of the aforementioned hash-tables and delivers them to Algorithm 14 for

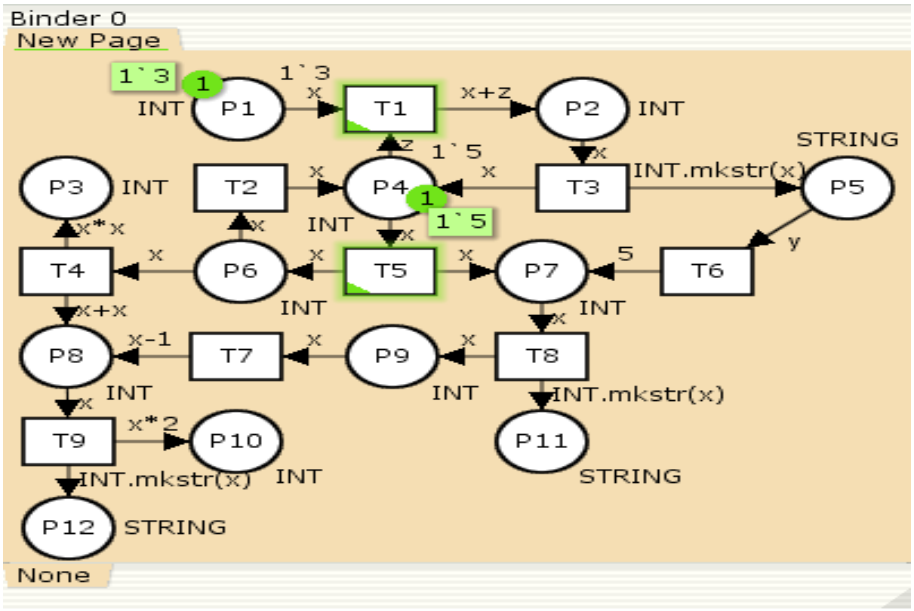


Figure 5.6: The example net for demonstrating the Lookup method.

attaching their adjoining vertices recursively. Thereupon at each stage the adjoining vertices of the components in a group are attached and these new components are compared to determine their similarity. Considering that the existing components were already compared in a previous step, the Lookup method compares these new components by comparing the last attached nodes. This continues until a component has no identical counterparts. Identifying components with multiple counterparts is beneficial because they will all be replaced with substitution transitions when a new module is constituted for the component. This forms the basis of our proposed reduction.

Phase 1: Creating and Populating the Hash-Tables

Determining identical vertices of a CPN model by storing them into appropriate index of a hash-table constitute the very first phase of the Lookup method. The index at which a vertex is stored depends on its indegree (number of input adjoining vertices) and outdegree (number of output adjoining vertices). Considering that the vertices at any particular index of a hash-table have the same values for indegree and outdegree, they are all structurally similar. The two disjoint sets of vertices in a CPN model (*places* and *transitions*) are stored into two separate hash-tables.

However, this phase has two basic prerequisites that need to be addressed. Primarily,

all places and transitions in the net must have unique names. This is essential in order to identify them in the hash table. Additionally, each vertex must store its indegree and outdegree. This is necessary to determine the index in the hash-table at which it would be stored. Therefore a "two-pass" method is appropriate for creating the hash tables, wherein the two basic prerequisites are fulfilled in "first pass" and thereupon the hash-tables are created in "second pass".

Steps for Creating and Populating Hash-Tables: Algorithm 12 lists the steps involved in creating and populating the two hash-tables. While the loop in steps 2-14 constitute the first pass, the other loop (steps 23-26) constitute the second pass. In the "first-pass", the algorithm needs to render a new name to each vertex and scrutinise them to determine their indegree and outdegree. In pursuit of fulfilling the first requirement, the algorithm uses two global identifiers, *placeId* and *transitionId*. As the names indicate, former is used for naming places while the latter is used for naming transitions. When a place is encountered during the first pass, the value stored in *placeId* is assigned as the name of the place as shown in step 4. This is followed by an increment of the value in *placeId* to ensure that the next place encountered is assigned a different name. Similarly, when a transition is encountered, its name is assigned using *transitionId* as shown in step 9. As the identifiers are initialised to '1' in step 1, the assigned names are integer values starting from 1. This is one possible scheme for assigning unique names and can be replaced by other possible schemes. In order to fulfil the other requirement, the *numInTran* and *numOutTran* properties of a vertex are incremented once for each of its input and output nodes (steps 6-7 and 11-12). Considering that these properties were initialised to '0' in steps 5 & 10, they record the indegree and outdegree of the corresponding places and transitions. The indegree and outdegree of a vertex can either be a positive integer or zero.

Once the prerequisites are fulfilled, the second pass can start creating the hash tables using the hash function defined in steps 17-22. It accepts a vertex and returns the index of the hash-table where it should be inserted. It calculates the index using the indegree and outdegree of the vertex that were determined during the first pass. Additionally, it also requires the total number of places and transitions in the net. These are determined in steps 15-16 and stored in identifiers *numPlace* and *numTran*. The hash function for a vertex v is

defined as

$$hashFun(v) = v.numInTran * (numTran + 1) + numOutTran//for\ a\ place \quad (5.1)$$

$$= v.numInPlace * (numPlace + 1) + v.numOutPlace//otherwise \quad (5.2)$$

where

- *v.numInTran, v.numOutTran*$\in$*[0,numTran]: v.numInTran + v.numOutTran*$\neq$*0*
- *v.numInPlace, v.numOutPlace*$\in$*[0,numPlace]: v.numInPlace + v.numOutPlace*$\neq$*0.*
- *v.numInXxxx* and *v.numOutXxxx* store the indegree and outdegree for a vertex v

These conditions ensure that isolated places or transitions are not processed. It is necessary to filter out isolated vertices as they have no adjoining nodes.

As discussed previously, the hash-tables are used to classify the vertices of a graph based on their indegree and outdegree. The legitimacy of the hash-function in Algorithm 12 depends on its ability to ensure that no two dissimilar vertices hash to the same index. This can be proved by demonstrating that any two arbitrary vertices v_1 and v_2 hashing to the same index must be structurally similar. Consider any two places p_1 and p_2 in a net with

indegree of p_1 = p_1*.numInTran=*x_1
outdegree of p_1 = p_1*.numOutTran=*y_1
indegree of p_2 = p_2*.numInTran=*x_2
outdegree of p_2 = p_2*.numOutTran=*y_2
total number of transitions = *numTran=z-1 (z*$\geq$*2)*

Since both v_1 and v_2 are places, their indices are determined using the hash-function *hash-Fun(v)* as defined in equation 5.1. Supposing both of these places hash to index I, we get

$$x_1 * z + y_1 = x_2 * z + y_2 = I \quad (5.3)$$

$$or \quad z * (x_1 - x_2) + (y_1 - y_2) = 0 \quad (5.4)$$

We know that the difference between two positive integers cannot be greater than the bigger integer. That is if $m \geq 0$, $n \geq 0$ are two integers such that m > n and $m+n \neq 0$, then

$$|m\text{-}n| \leq m \quad (5.5)$$

Since $0 \leq y_1, y_2 < z$ and $0 \leq x_1, x_2 < z$, it can be deduced from equation 5.5 that

$$0 \leq |y_1 - y_2| < z \qquad and \qquad 0 \leq |x_1 - x_2| < z \tag{5.6}$$

where $|y|$ is the modulus function that returns the absolute value of an enclosed variable or expression. Although $|x_1 - x_2| \in [0, z)$, we find it particularly interesting when $|x_1 - x_2| \geq 1$. Multiplying the positive integer z on both sides, we get

$$z * |x_1 - x_2| \geq z \tag{5.7}$$

when $|x_1 - x_2| \geq 1$. However, since we deduced $|y_1 - y_2| < z$ in equation 5.6, the first term in equation 5.4 is greater than second term and cannot cancel it to produce zero. Consequently $|x_1 - x_2| < 1$ and equation 5.4 only holds for $|x_1 - x_2| = 0$ (as $|x_1 - x_2|$ cannot be a fraction). This imparts $x_1 = x_2$ and using this result in equation 5.4, we get $y_1 = y_2$. The two places p_1 and p_2 are therefore structurally similar, demonstrating that any two vertices hashing to the same index must be identical. The proof also applies when p_1 and p_2 are transitions. The potency of the hash function is thereby established.

Figure 5.7 shows the model in Figure 5.6 after first-pass wherein each vertex is assigned a unique name and its degree determined. The vertices are represented using colour codes wherein all vertices marked with same colour have the same values for indegree and outdegree and are therefore structurally similar. For instance all red spots correspond to indegree of 1 and outdegree of zero. Similarly, a black spot in a transition corresponds to indegree of one and outdegree of two. Furthermore in order to avoid confusion, we have appended a 'P' or 'T' to places and transitions names. For instance a place named '1' in the first-pass is called 'P1' in Figure 5.7 while a transition named '1' is called 'T1'.

Thereupon the second pass populates the hash-tables for places and transitions based on their indegrees and outdegrees. The places and transitions are hashed into separate hash-tables known as *hashPlace* and *hashTran* as shown in steps 24-25. The hash-tables created for this model in the second-pass are shown in Tables 5.2 and 5.3. The first column in these tables indicate the index of the hash-table where the vertices were inserted. At the conclusion of second pass, the smallest problem of finding all structurally similar vertices in a CPN (i.e. the graph based model) is solved. The Lookup method being a 'Decrease-and-Conquer' algorithm, the results obtained in this phase are used in the next phase for solving a bigger problem. Consequently, the hash-tables are sent to Algorithm 13 for further processing.

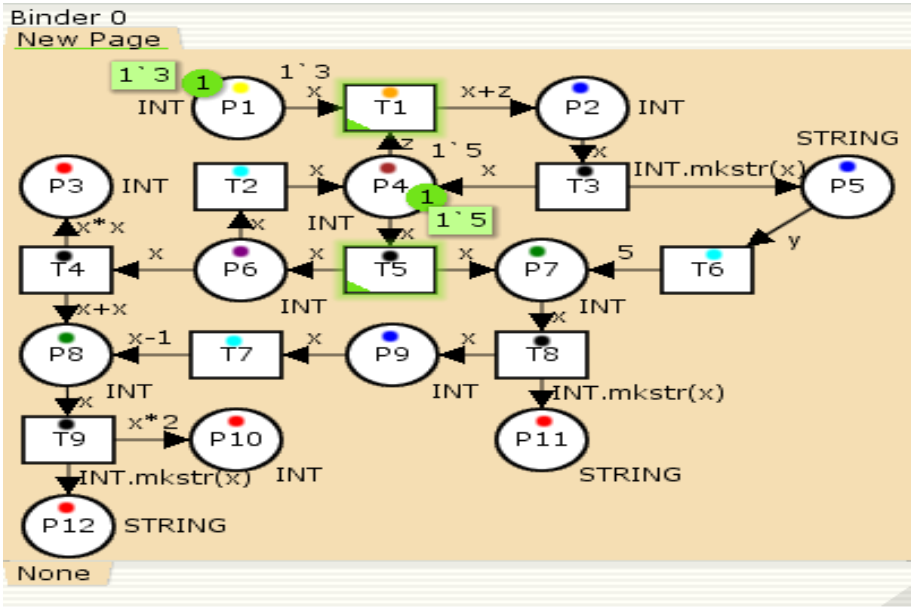


Figure 5.7: The example net after Phase-1.

Phase 2: Recursively Adding Vertices

As demonstrated previously, the hash-function *hashFun(v)* defined in equations 5.1 & 5.2 ensure that all vertices at an index of a hash-table are structurally similar. Such similar vertices form a group, known as *elementary group*.

Definition 15 *A group is defined as a maximal set of non-overlapping components in a graph (or graph based model) that are all structurally similar.*

Corollary 8 *Two components of a graph are non-overlapping if they do not share any vertices.*

Corollary 9 *For a graph N, $\alpha_N = \{G_1, G_2, \cdots, G_g\}$ denotes the set of groups in it.*

Corollary 10 *A group $G_i \in \alpha_N$ is a maximal set if $\forall j \in [1,g]$: $(i \neq j) \wedge (G_i \not\subseteq G_j)$.*

Corollary 11 *A group consisting of only single vertex is called elementary group.*

Corollary 12 *The set of groups α_N for a graph N is not unique for a graph.*

Figure 5.8 demonstrates a possible set of groups in our example CPN. The groups in this set are listed in equation 5.8. While the blue and red vertices in this figure form elementary

Algorithm 12: CreateHashTables(CPN N)

```
   Data: CPN N
   Result: Hash tables are generated
 1 placeId←transitionId←1;
 2 foreach Vertex V in N do //First Pass
 3 |  if V is a Place then
 4 |  |  V.name←placeId++;                          // Assign unique name
 5 |  |  V.numInTran←V.numOutTran←0;                       // initialise
 6 |  |  foreach input transition T_in of V do V.numInTran++;  // Find indegree
 7 |  |  foreach output transition T_out of V do V.numOutTran++;  // Find
   |  |  outdegree
 8 |  else if V is a Transition then
 9 |  |  V.name←transitionId++;                     // Assign unique name
10 |  |  V.numInPlace←V.numOutPlace←0;                     // initialise
11 |  |  foreach input place P_in of V do V.numInPlace++;  // Find indegree
12 |  |  foreach output Place P_out of V do V.numOutPlace++; // Find outdegree
13 |  end
14 end
15 numPlace=placeId-1;                    // total number of Places in net N
16 numTran=transitionId-1;              // total number of transitions in net N
17 fun hashFun(v)= switch v do //Define hash function
18 |  case v is a Place
19 |   return v.numInTran*(numTran+1)+v.numOutTran;
20 |  otherwise
21 |   return v.numInPlace*(numPlace+1)+v.numOutPlace;
22 end
23 foreach Vertex V in N do //Second Pass
24 |  if V is a Place then hashPlace[hashFun(V)]=V;     // hash-table for place
25 |  if V is a Transition then hashTran[hashFun(V)]=V;  // hash-table for tran
26 end
```

groups, the set of brown vertices connected with brown arcs constitute identical components. The notation *node(list of input vertices: list of output vertices)* is used to illustrate the input and output vertices of a node.

$$\alpha_{example} = \{\{P3, P10, P11, P12\}, \{P2, P5, P9\}, \{P8(T4, T7 : T9), P7(T5, T6 : T8)\}\} \quad (5.8)$$

Table 5.2: Hash-table for places created in 2^{nd} pass

Index	H(in,out)	Colour	List of places(in:out Transition)
1	H(0,1)	Yellow	P1(T1)
10	H(1,0)	Red	P11(T8),P12(T9),P10(T9),P3(T4)
11	H(1,1)	Blue	P2(T1:T3),P5(T3:T6),P9(T8:T7)
12	H(1,2)	Purple	P6(T5:T4,T2)
21	H(2,1)	Green	P7(T5,T6:T8),P8(T4,T7:T9)
22	H(2,2)	Brown	P4(T2,T3:T1,T5)

Table 5.3: Hash-table for transitions created in 2^{nd} pass

Index	H(in,out)	Colour	List of transitions(in:out Places)
14	H(1,1)	Cyan	T2(P5:P3),T6(P4:P6), T7(P8:P7)
15	H(1,2)	Black	T3(P1:P3,P4),T4(P5:P2,P7), T5(P3:P5,P6), T8(P6:P8,P10), T9(P7:P9,P11)
27	H(2,1)	Orange	T1(P3,P12:P1)

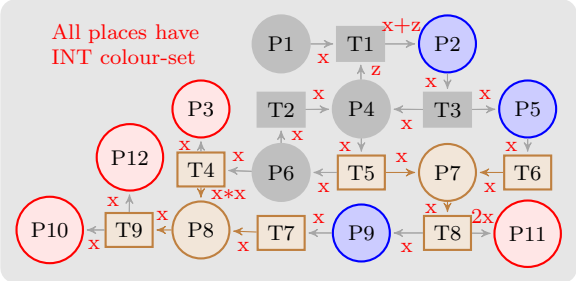


Figure 5.8: A possible set of groups in $\alpha_{example}$ for our example CPN.

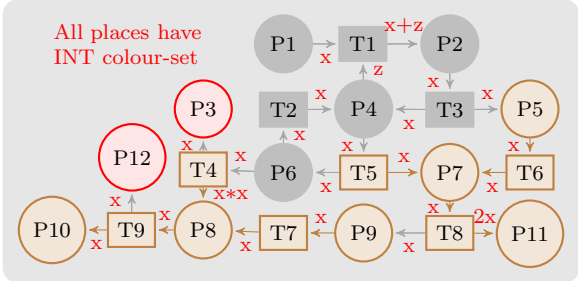


Figure 5.9: α is not unique. Another possible set of groups in $\alpha_{example}$ for same CPN.

However, from Corollary 12, α is not unique for a graph based model. Figure 5.9 demonstrates another possible set of groups in the example CPN and these are listed in equation 5.9.

$$\alpha_{example} = \{\{P3, P12\}, \{P8(T4, T7(P9) : T9(: P10)), P7(T5, T6(P5) : T8(P11))\}\} \quad (5.9)$$

Since at this point the smallest problem has been solved, the results are available for the aforementioned bottom-up resolution of larger problems. This is handled by the 2^{nd} phase of the Lookup method that rolls in after the first phase populates the hash-tables.

This phase has two related sub-phases. The first sub-phase, the steps for which are listed as Algorithm 13, renders all elementary groups of places from the hash-tables. Considering that any particular index of a hash-table contains all identical vertices, this sub-phase in-effect fetches the vertices at each index to constitute elementary groups. Each of these elementary groups are thereupon forwarded to the other sub-phase (listed as Algorithm 14) for recursively attaching the adjoining vertices and comparing them to determine similarity. Attaching adjoining vertices allow determining components of larger size.

Why Use Additional Hash-Tables? Before passing the elementary groups to the second sub-phase, some additional information is stored into them in the first sub-phase. For efficiency and convenience, this information is stored in two levels of hash tables. In this section, the information that is stored and the reason for storing this information is discussed using Figure 5.10. An insight into this would ensure better understanding of the two sub-phases and the related algorithms introduced later.

Consider an elementary group 'G', shown in Figure 5.10, that consists of five places

$$G=\{P1, P2, P3, P4, P5\}$$

The ordering of these places is crucial and cannot be changed at a later stage because of the underlying data-structures used. Each place in a group contains the two hash-tables known as *inCompare* and *outCompare*. For a place P, these hash-tables are referred as *P.inCompare* and *P.outCompare* in Figure 5.10. The keys in these hash-tables consist of places that are on the right-hand side of P in its group. For instance in Figure 5.10, the hash-tables for P2 have {P3, P4 and P5} as its keys because these places happen to be at P2's right in group G. Hereafter the position of these places needs to be maintained in the group to ensure the validity of entries in these hash-tables. The value corresponding to a key consists of a pointer to another hash-table as shown using the blue arrows in Figure 5.10. The significance of these hash-tables (hashIn_{ij} and hashOut_{ij}) is explained using Figure 5.11

that consists of two identical components (shown in brown) obtained by adding the adjoining transitions of P7 and P8. Because these components are identical, each transition in a component has an identical counterpart in the other component (T4↔T5, T6↔T7 and T8↔T9 from Table 5.3). The next step involves attaching the adjoining places of these transitions to obtain even larger components. However, in order to ensure that the components are similar even after adding these places, we need to compare the adjoining places of each transition with that of its counterpart prior to attaching them. These hash-tables map similar vertices between two components in order to help us in making this comparison.

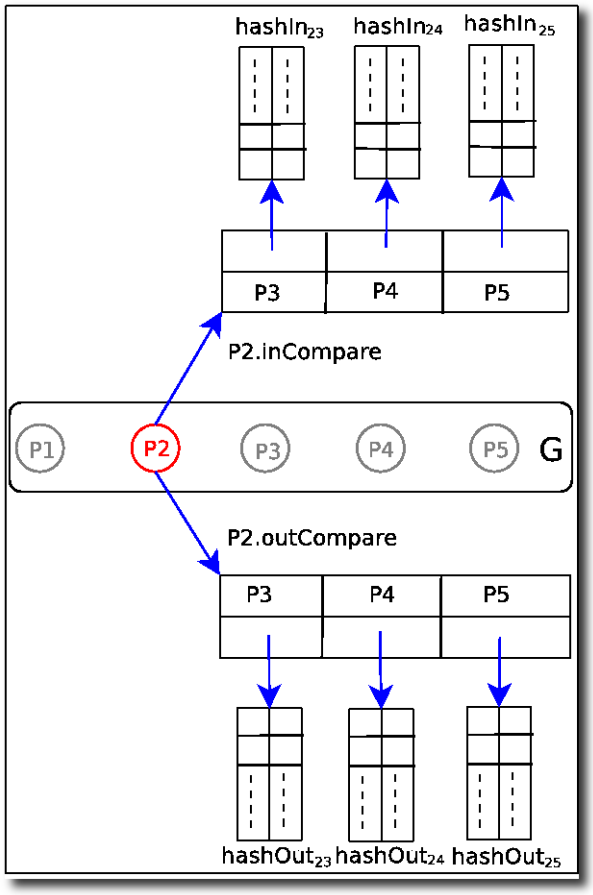


Figure 5.10: The hash-tables corresponding to a place.

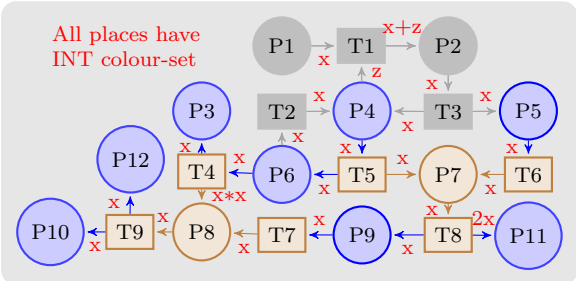


Figure 5.11: The adjoining places of identical components to be added next

For instance P7.inCompare[P8] will have two entries {(T6,T7),(T5,T4)} whose input places need to be compared[2] while P7.outCompare[P8] will also have two entries {(T8,T9),(T5,T4)} whose output places need to be compared[3]. Note that (T5,T4) appears in both tables because both its input and output places need to be compared. Moreover, P7 and P8 do not appear in any of these tables in spite of being similar because they do not have adjoining vertices that need to be compared.

In order to further elaborate on the necessity of these hash-tables, consider the components formed after adding the adjoining transitions of any two places P2 and P4 as shown in Figure 5.12. For determining the structural similarity of these components we need to first find the hash-tables hashIn and hashOut.

$$\text{hashIn}_{24} = \text{P2.inCompare[P4] and hashOut}_{24} = \text{P2.outCompare[P4]}$$

Considering that the similarity of these two components can only be determined by comparing the input and output transitions of places P2 and P4 (and not any other vertices), both the hash-tables should contain a single entry (P2,P4) as key-value pair and are shown in Tables 5.4 and 5.5.

This being a trivial case, the hash-tables were initialised with these entries when they were created. Hereafter, the entries in these hash-tables would be updated after each successful comparison. For instance if the components in Figure 5.12 are found to be similar, the previous entries in the hash-table would be deleted and new entries would be added. The new entries will depend on how the transitions in a component could be mapped to their structurally similar counterparts in other components. If this mapping corresponds to the shades as shown in Figure 5.13, the new entries would be as shown in Tables 5.6 and 5.7.

The usefulness of these hash-tables could be further illustrated by comparing larger components. This is accomplished by comparing the components C1′ and C2′ in Figure 5.13 that are formed after appending the adjoining places to components C1 and C2. The similar components C1 and C2 are bordered in red in Figure 5.13 while their adjoining places are bordered in blue. In order to decide on the usefulness of the hash-tables, we first try to compare C1′ and C2′ without using them. Considering that C1 and C2 were already found similar, only their adjoining places need to be compared to determine the similarity of C1′ and C2′. However, there is no way to decide on a place in the first component that should be compared with a place in second component. For instance we do not know the place in sec-

[2] i.e. $\forall(v_1, v_2) \in P_x.inCompare[P_y]$, compare input vertices of v_1 and v_2

[3] i.e. $\forall(v_1, v_2) \in P_x.outCompare[P_y]$, compare output vertices of v_1 and v_2

Table 5.4: Initial $hashIn_{24}$

Key	Value
P2	P4

Table 5.5: Initial $hashOut_{24}$

Key	Value
P2	P4

Table 5.6: $hashIn_{24}$ after adding the adjoining transitions

Key	Value
T1	T5
T3	T6

Table 5.7: $hashOut_{24}$ after adding the adjoining transitions

Key	Value
T2	T4

ond component to which R1 should be compared. Similarly, the place in the first component to which R8 should be compared to is unknown. The hash-tables returned by inCompare and outCompare provide the additional information required for undertaking these decisions. The entry (T1,T5) in hashIn prompts us to compare the input places of T1 with those of T5. Similarly an entry (T2,T4) in hashOut prompt us to compare the output places for T2 and T4. The benefits that justify using these additional hash-tables are:

1. For any two components *C1* and *C2*, the hash-tables *inCompare* and *outCompare* are needed to fetch the secondary hash-tables $hashIn_{ij}$ and $hashOut_ij$.

2. The secondary hash-tables are needed to fetch the vertices whose input and output nodes need to be compared to determine the similarity of *C1* and *C2*.

Sub-Phase 1: Finding All Elementary Group of Places: In a decrease-and-conquer strategy, the solution for a smaller problem is used to solve a larger problem. After addressing the smallest problem of finding identical vertices in a model, we use the results to determine identical components of bigger size. This phase in effect fetches the vertices from each index of the hash-tables populated in the first phase and forwards them to the 2^{nd} sub-phase. However, as discussed in previous section, additional hash-tables are initialised for each vertex before forwarding.

Algorithm 13 lists the steps for finding all elementary group of places. All places at an index of hashPlace are retrieved and stored in list P as illustrated in step 2. The places in list P constitute an elementary group, wherein the name of a component is the same as that of its sole constituent place. Each place *pl* in P is added to *pl.iList* and *pl.oList* (step 5) and later used by Algorithm 14 to create a hash-table for indices. Thereafter the algorithm initialises the hash-tables hashIn_{ij} and hashOut_{ij} with (P_i,P_j) in steps 8-9 as their adjoining transitions

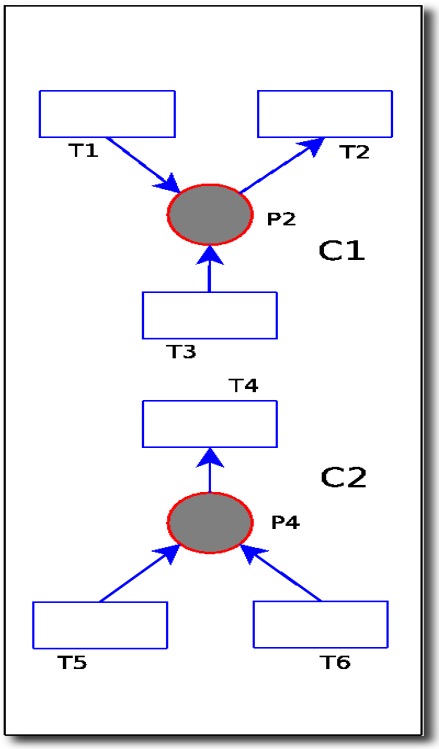


Figure 5.12: After adding the adjoining transitions to P2 and P4

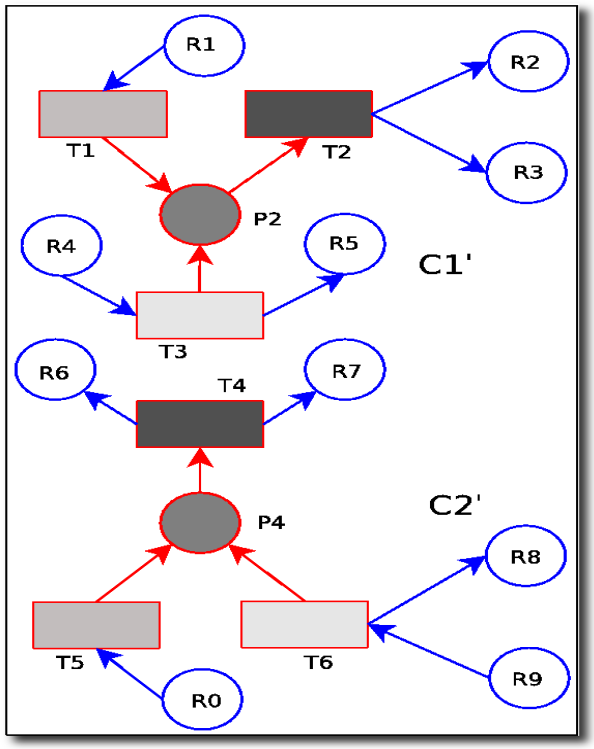


Figure 5.13: After adding the adjoining places to component in Figure 5.12

need to be compared next, to ascertain the equality of components. These hash-tables are stored in Pi.inCompare[Pj] and Pi.outCompare[Pj] in steps 10-11.

This being a decrease-and-conquer algorithm, the results obtained in this step are used in the next step for solving a bigger problem. The bigger problem in this case is finding larger identical components. In this pursuit, the list of places P along with all additional information is passed to Algorithm 14 for adding adjoining vertices and forming new groups.

Sub-Phase 2: Recursively Determining Larger Groups: Algorithm 14 lists the steps for creating a meta-group from a group by augmenting each component with vertices from its immediate vicinity. As contrary to group, a meta-group consists of components that are not essentially similar. After creating a meta-group, the algorithm compares the components in it to forge new groups.

Definition 16 *A meta-group is defined as a maximal set of non-overlapping components of a graph which are not necessarily similar.*

Corollary 13 *A meta-group should always be reducible to a group by removing the most recently added adjoining vertices.*

Corollary 14 *Elementary meta-group does not exist because it cannot be reduced to a group.*

Algorithm 13: findElementaryGroups(hashPlace)

```
   Data: Hash Table For Places
   Result: Elementary group of places
 1 foreach index i in hashPlace do
 2 |   list P←hashPlace[i] ;                  // places at index i are copied to P
 3 |   for i←1 to P.count() do //P.count gives number of places in P
 4 |   |   place pl=P.at(i);                  // get place at position i of list P
 5 |   |   pl.iList=pl.oList=pl;
 6 |   |   for j←i+1 to P.count() do
 7 |   |   |   place pl2←P.at(j);
 8 |   |   |   hashIn_ij[pl]←pl2;
 9 |   |   |   hashOut_ij[pl]←pl2;
10 |   |   |   pl.inCompare[pl2]=hashIn_ij;
11 |   |   |   pl.outCompare[pl2]=hashOut_ij;
12 |   |   end
13 |   end
14 |   Group(P);                                  // add transitions and regroup
15 end
```

After determining the elementary group of places, Algorithm 13 delivers them to Algorithm 14 one after the other. The latter stores each group into list V before processing it. If a group has only a single component, the algorithm terminates at the very first step without any processing. This is consistent with the objective of the Lookup method to determine groups with multiple identical components.

Why Use Additional Lists and Hash-Tables?: In an earlier discussion, we had introduced the hash-tables *inCompare*, *outCompare*, *hashIn* and *hashOut*. These hash tables specifically contain the vertices that need to be compared to determine the similarity of two components. This section introduces lists that are used to store the vertices whose adjoining nodes are to be added to the components in the next recursion of Algorithm 14. For a component C, these lists are stored in C.iList and C.oList. As the names indicate, the input vertices for all $v_i \in$C.iList and output vertices for all $v_o \in$C.oList are added to C in the next recursion. In addition, the hash-table hashInIndex and hashOutIndex are used to store the indices of the vertices to be added in the next step. As explained later, these

indices are calculated using the hash-function in Algorithm 12 and they help in comparing the components for similarity.

Algorithm 14 stores each elementary group of places into V before processing it. As mentioned earlier, each place $E \in$V has a pair of lists (i.e. *E.iList* and *E.oList*) that were assigned to E by Algorithm 13. Consequently this ensures that the adjoining vertices of E are attached to it in the next recursion. For a place P2∈V, these lists are illustrated using Figure 5.14. Thereupon the vertices in these lists are fetched by Algorithm 14 to determine the indices of their input and output nodes. The nested *for* loop in steps 3-8 is responsible for determining the indices of input vertices for each node in *iList* and subsequently storing them into the hash-table *hashInIndex*. Similarly the nested loop in steps 9-14 determine the indices of output vertices for each node in *oList* before storing them into the hash-table *hashOutIndex*. For simplicity, the algorithm do not process the output vertices for the nodes in *iList* and the input vertices for the nodes in *oList*. The place P2 in Figure 5.14 has two input transitions T1 and T3 and an output transition T2. Assuming these transitions occupy the following indices in the hash-tables that were populated in the first phase,

hashFun(T1)=I1
hashFun(T2)=I2
hashFun(T3)=I3

the entries (P2,I1) & (P2,I3) would be added to hashInIndex while a single entry (P2,I2) would go into *hashOutIndex* as shown in Figure 5.14. As a result, *hashInIndex*[P2] would return the list of indices corresponding to all input vertices of P2 (i.e. [I1,I2]). It should be noted that the hash-tables *hashInIndex* and *hashOutIndex* can contain multiple entries for the same key value. Furthermore these hash tables have a single instance each to store the indices for all the vertices. After the loop terminates in step 16, each input node for the vertices in E.iList and each output node for the vertices in E.oList are considered 'added' to the subnet wherein all the subnets together form a meta-group. The vertices in these lists are thereupon replaced by their input and output nodes. This ensures that the next set of adjoining vertices are attached in the following recursion of Algorithm 14. For example place P2 is deleted from P2.iList and replaced by its input transitions T1 and T3. Similarly P2 is deleted from P2.oList and replaced with its sole output transition T2 as illustrated in Figure 5.15.

As evident from Figures 5.14 and 5.15, the hash-tables hashInIndex and hashOutIndex are updated on every recursion of Algorithm 14. In the next recursion of Algorithm 14, the

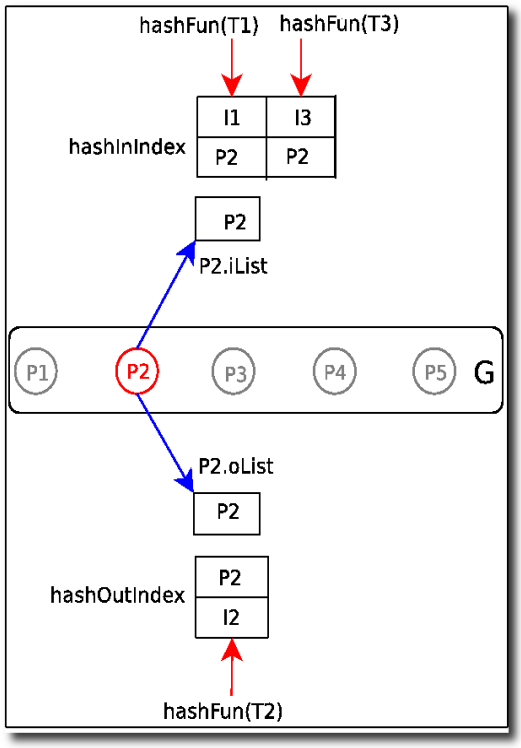


Figure 5.14: The key-value pairs inserted into the hash-tables hashInIndex and hashOutIndex in first execution of Algorithm 14.

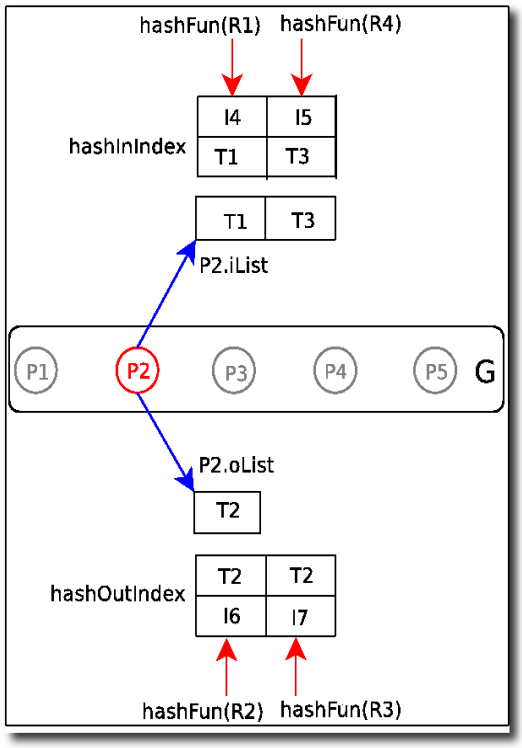


Figure 5.15: The key-value pairs inserted into the hash-tables hashInIndex and hashOutIndex in second execution of Algorithm 14.

entries added to hashInIndex depend on the indices of input places for transitions in P2.iList. Similarly the entries added to hashOutIndex would depend on the indices of output places for transitions in P2.oList. Supposing that the component formed after attaching T1, T2 and T3 to P2 is similar to C1′ in Figure 5.13 and the adjoining places R1, R2, R3 and R4 to be added occupy the following indices in the hash-table for places determined in the first phase,

$$\begin{aligned}\text{hashFun(R1)}&=\text{I4}\\ \text{hashFun(R2)}&=\text{I6}\\ \text{hashFun(R3)}&=\text{I7}\\ \text{hashFun(R4)}&=\text{I5}\end{aligned}$$

these hash-tables would store entries as shown in Figure 5.15.

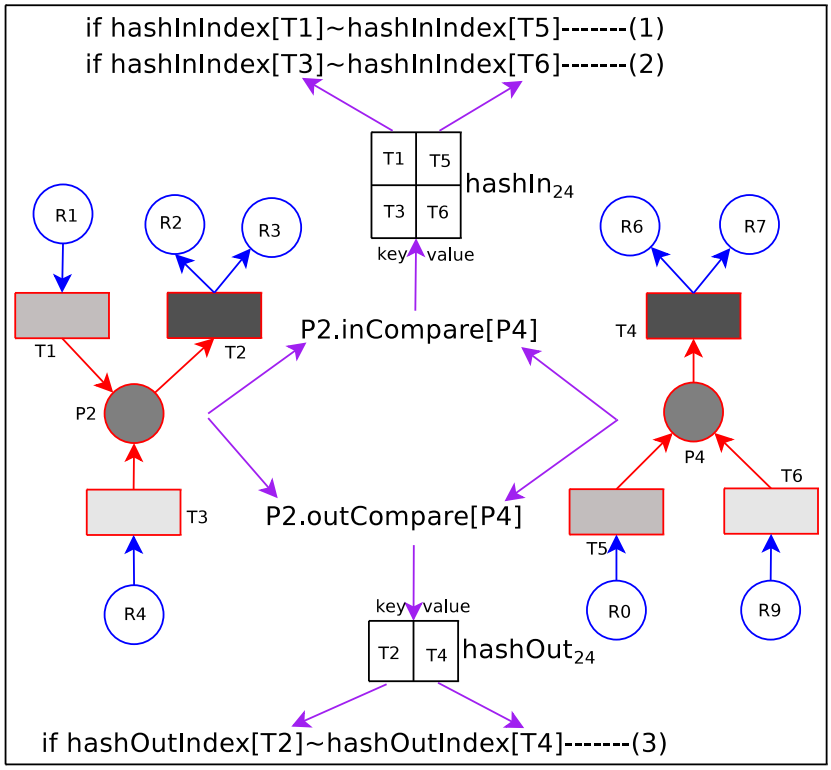


Figure 5.16: Comparing two components using Algorithm 14.

It is worth mentioning that for any component E in list V, vertex E is not same as component E. If E is at position i in V, vertex E refers to place P_i in corresponding elementary group. On the contrary component E refers to place P_i along with all adjoining nodes attached to it by Algorithm 14. For instance in Figure 5.13, vertex C1′ refer to place P2 while vertex C2′ refer to place P4. Furthermore even if E is a component, it is implicitly vertex E when referring to E.iList or E.inCompare.

Thereafter the *foreach* loop in steps 17-44 compare the subnets in this meta-group to formulate the new groups. For any two components E and F in V, the hash-tables hashIn and hashOut are fetched from E.inCompare[F] and E.outCompare[F] in steps 24-25. This is illustrated in Figure 5.16 for two components formed out of places P2 and P4. As manifested previously, E.inCompare[F] and E.outCompare[F] are defined only if E appears prior to F in V. If Algorithm 14 is processing an elementary group sent by Algorithm 13, both these hash-tables contain a single key-value pair (pl_i,pl_j). Otherwise as shown in Figure 5.16, these hash-tables will contain vertices whose input and output nodes need to be compared to determine similarity of the components. The meta-group MGr in Figure 5.16 and the group

G it could be reduced to are as follows

$$\text{MGr}=\{\text{P2(T1(R1),T3(R4):T2(:R2,R3)),P4(T5(R0),T6(R9):T4(R6,R7))}\}=\{\text{K1}',\text{K2}'\}$$
$$\text{Gr}=\{\text{P2(T1,T3:T2),P4(T5,T6:T4)}\}=\{\text{K1,K2}\}$$

The structural similarity of components in meta-group MGr can be determined by comparing the places added to K1 and K2 to form K1′ and K2′. In other words, the similarity of K1′ and K2′ could simply be determined by comparing the input vertices for each key-value pair in hashIn and output vertices for each key-value pair in hashOut. For example in Figure 5.16 the two components can be compared by comparing

1)Input places of T1 with those of T5
2)Input places of T3 with those of T6
3)Output places of T2 with those of T4

If these places are found to be structurally similar, these components are also determined to be similar and together form a group.

However, instead of comparing two vertices for structural similarity, we compare their indices in the corresponding hash-table. If two places occupy the same indices in the hash-table for places, they are deemed to be identical. This follows from an earlier demonstration that if hashFun(v_1)=hashFun(v_2), the vertices v_1 and v_2 must be structurally identical (equations 5.4 to 5.7). We have mentioned that hashInIndex stores the indices for all input vertices for a node while hashOutIndex stores the indices for all output vertices for a node. Therefore hashInIndex[T1] would give the set corresponding to indices of its input places. If this set is found to be similar to the set returned by hashInIndex[T5], we can conclude that each input place in T1 has a structurally similar counterpart that is an input place of T5.

Additionally if we determine the following:

$$\text{hashInIndex[T1]}\sim\text{hashInIndex[T5]}$$
$$\text{hashInIndex[T3]}\sim\text{hashInIndex[T6]}$$
$$\text{hashOutIndex[T2]}\sim\text{hashOutIndex[T4]}$$

we can conclude that K1′ and K2′ are structurally similar. A new group can be constituted using K1′ and K2′ into which other identical subnets are added after a similar comparison.

Later this new group is handed over to Algorithm 14 for further processing in step 43. The symbol $\sim$ is used to represent the similarity of two sets. Two lists are considered similar if they contain same number of elements and one is a permutation of other.

Algorithm 14: Group(list V)

```
   Data: List of vertices V
   Result: β_N, the superset groups of N
 1 if numberOfElements(V)=1 then return // List has one item
 2 foreach Vertex E in V do
 3     for i←1 to E.iList.count() do
 4         Vertex D←E.iList.at(i);                                  // Get the vertex at position i
 5         foreach Vertex C in D.inVertices() do hashInIndex[D]←hashFun(C);
 6         newInList.append(D.inVertices());                        // copy input vertices
 7         E.iList.remove(D);                                       // remove this vertex from E.iList
 8     end
 9     for i←1 to E.oList.count() do
10         Vertex D←E.oList.at(i);                                  // Get the vertex at position i
11         foreach Vertex C in D.outVertices() do hashOutIndex[D]←hashFun(C);
12         newOutList.append(D.outVertices());                      // copy output vertices
13         E.oList.remove(D);                                       // remove the vertex from E.oList
14     end
15     E.iList←newInList; E.oList←newOutList;                       // copy vertices
16 end
17 foreach i←1 to V.count() do //form groups out of meta-group V
18     E←V.at(i);                                                   // get the vertex at i
19     list newV.append(E);                                         // add it to new list
20     V.remove(E);                                                 // remove it from V
21     flag=0;                                                      // a flag to find equality
22     foreach j←i+1 to V.count() do //E appear prior to F in V
23         F←V.at(j);                                               // get vertex at j
24         hashIn←E.inCompare[F];                                   // get the hash-tables containing
25         hashOut←E.outCompare[F];                                 // i/o vertices to be compared
26         foreach V_in1, V_in2, V_out1, V_out2:hashIn[V_in1]=V_in2 & hashOut[V_out1]=V_out2 do
27             list inIndex1←hashInIndex[V_in1];
28             list inIndex2←hashInIndex[V_in2];
29             list outIndex1←hashOutIndex[V_out1];
30             list outIndex2←hashOutIndex[V_out2];
31             hashIn.remove(V_in1); hashOut.remove(V_out1);
32             if (inIndex1≁inIndex2) or (outIndex1≁outIndex2) then
33                 flag←1; break;
34             else
35                 foreach I_1 ∈V_in1.inVertices & O_1 ∈V_out1.outVertices do
36                     if I_2 ∈ V_in2.inVertices & hashFun(I_1)=hashFun(I_2) then hashIn[I_1]←I_2;
37                     if O_2 ∈V_out2.outVertices & hashFun(O_1)=hashFun(O_2) then hashOut[O_1]←O_2;
38                 end
39             end
40         end
41         if flag=0 then {newV.append(F); V.remove(F);}
42     end
43     Group(newV);                                                 // add vertices and regroup
44 end
```

Table 5.8: Initialisation of hashInIndex is followed by a change in iList for each subnet in M_1

Initial inList	hashInIndex Initialisation	Updated inList
P11.inList={P11}	hashInIndex[P11]={hashFun(T8)}={2}	P11.inList={T8}
P12.inList={P12}	hashInIndex[P12]={hashFun(T9)}={2}	P12.inList={T9}
P10.inList={P10}	hashInIndex[P10]={hashFun(T9)}={2}	P10.inList={T9}
P3.inList={P3}	hashInIndex[P3]={hashFun(T4)}={2}	P3.inList={T4}

Determining Similarity in the Example Net

We now consider applying Algorithms 13 and 14 to our example net in Figure 5.6. The elementary groups are fetched by Algorithm 13 and their additional hash-tables are initialised before passing them to Algorithm 14. The set of places {`P11`, `P12`, `P10`, `P3`} are all at index 2 of table 5.2 and consequently forms an elementary group V_1. For any two places pl_i, $pl_j \in V_1$: pl_i appears before pl_j in V_1, pl_i.inCompare[pl_j] is assigned an hash-table that is initialised with a single entry (pl_i, pl_j). This is illustrated in Figure 5.17 for each pair of places in V_1. Additionally, for each place pl$\in V_1$, the lists pl.iList and pl.oList are initialised with a single entry pl. Subsequently Algorithm 13 passes V_1 to Algorithm 14 for further processing.

Algorithm 14 attaches each place pl$\in V_1$ to its sole input transitions and produces a meta-group

$$M_1=\{\texttt{P11}\leftarrow\texttt{T8}, \texttt{P12}\leftarrow\texttt{T9}, \texttt{P10}\leftarrow\texttt{T9}, \texttt{P3}\leftarrow\texttt{T4}\}$$

as shown in Figure 5.17. In this figure, the transitions attached are shown in blue. The index for each input transition in iList is found and stored in hashInIndex. This is done as explained earlier using Figures 5.14 and 5.15. The other list oList is empty owing to zero output transitions. The values added to hashInIndex and the corresponding changes to inList are listed in Table 5.8. Subsequently the meta-group members need to be compared to establish new groups consisting of identical subnets. For any two places pl_i, $pl_j \in V_1$: pl_i appears before pl_j in V_1, the components formed out of pl_i and pl_j can be compared by obtaining hash-table $hashIn_{ij}$(=pl_i.inCompare[pl_j]) and checking if hashInIndex[v1]$\sim$hashInIndex[v2] for each entry (v1,v2) in hashIn. These hash-tables are shown in Figure 5.17. Considering that hashInIndex returns similar sets for each pair of vertices in hash-table hashIn,

$$\text{hashInIndex[P3]}\sim\text{hashInIndex[P10]}\sim\{2\}$$
$$\text{hashInIndex[P3]}\sim\text{hashInIndex[P11]}\sim\{2\}$$
$$\text{hashInIndex[P3]}\sim\text{hashInIndex[P12]}\sim\{2\}$$

Table 5.9: Initialisation of hashInIndex is followed by a change in iList for each subnet in M_2

Initial inList	hashInIndex Initialisation	Updated inList
P11.inList={T8}	hashInIndex[T8]={hashFun(P7)}={5}	P11.inList={P7}
P12.inList={T9}	hashInIndex[T9]={hashFun(P8)}={5}	P12.inList={P8}
P10.inList={T9}	hashInIndex[T9]={hashFun(P8)}={5}	P10.inList={P8}
P3.inList={T4}	hashInIndex[T4]={hashFun(P6)}={4}	P3.inList={P6}

$$\text{hashInIndex[P10]}\sim\text{hashInIndex[P11]}\sim\{2\}$$
$$\text{hashInIndex[P10]}\sim\text{hashInIndex[P12]}\sim\{2\}$$
$$\text{hashInIndex[P11]}\sim\text{hashInIndex[P12]}\sim\{2\}$$

the meta-group has all identical subnets and the new group V_2 is equivalent to meta-group M_1, or

$$V_2=M_1=\{\texttt{P11}\leftarrow\texttt{T8}, \texttt{P12}\leftarrow\texttt{T9}, \texttt{P10}\leftarrow\texttt{T9}, \texttt{P3}\leftarrow\texttt{T4}\}.$$

This group is shown in Figure 5.18 wherein it is encircled in red for easy identification. The corresponding shades indicate identical places and transitions. Each hash-table hashIn is modified to contain an entry for the corresponding pair of transitions found similar. These modified hash-tables are also illustrated in Figure 5.18. V_2 is then sent back to Algorithm 14 for further processing.

The above manifested process is then repeated when Algorithm 14 attaches input places to each subnet S $\in V_2$. Consequently, a new meta-group

$$M_2= \{\texttt{P11}\leftarrow\texttt{T8} \leftarrow\texttt{P7}, \texttt{P12}\leftarrow\texttt{T9}\leftarrow\texttt{P8}, \texttt{P10}\leftarrow\texttt{T9}\leftarrow\texttt{P8}, \texttt{P3}\leftarrow\texttt{T4}\leftarrow\texttt{P6}\}$$

is obtained as shown in Figure 5.18. For each transition T$\in$ S.iList, indices of its input places are found and inserted into hashInIndex[T] as shown in Table 5.9. Thereupon the subnets in meta-group are compared to establish groups. For any two components E, F$\in V_2$ formed out of places pl_i, pl_j $\in V_1$:pl_i appears before pl_j in V_1, the similarity of E and F can be determined by obtaining hash-table $hashIn_{ij}$(=pl_i.inCompare[pl_j]) and checking if hashInIndex[v1]=hashInIndex[v2] for each entry (v1,v2) in hashIn. These hash-tables are shown in Figure 5.18. Comparing the sets returned by hashInIndex for each pair of vertices in hash-table hashIn,

$$\text{hashInIndex[T4]}\not\sim\text{hashInIndex[T9]}$$
$$\text{hashInIndex[T4]}\not\sim\text{hashInIndex[T8]}$$

Table 5.10: Initialisation of hashInIndex is followed by a change in iList for each subnet in M_3

Initial inList	P10.inList={P8}
hashInIndex Initialisation	hashInIndex[P8]={hashFun(T4), hashFun(T7)}={2, 1}
Updated inList	P10.iList={T4, T7}
Initial inList	P11.inList={P7}
hashInIndex Initialisation	hashInIndex[P7]={hashFun(T5), hashFun(T6)}={2, 1}
Updated inList	P11.iList={T5, T6}

$$\text{hashInIndex[T4]} \not\sim \text{hashInIndex[T9]}$$
$$\text{hashInIndex[T9]} \sim \text{hashInIndex[T8]} \sim \{5\}$$
$$\text{hashInIndex[T9]} \sim \text{hashInIndex[T9]} \sim \{5\}$$
$$\text{hashInIndex[T8]} \sim \text{hashInIndex[T9]} \sim \{5\}$$

only the first three subnets in M_2 are found to be similar. However, the 2^{nd} and 3^{nd} subnets are overlapping and cannot be together in the same group. Consequently the new group is

$$V_3=\{\texttt{P11}\leftarrow\texttt{T8}\ \leftarrow\texttt{P7},\ \texttt{P10}\leftarrow\texttt{T9}\leftarrow\texttt{P8}\}$$

and is illustrated in Figure 5.19 using red-borders. The figure also demonstrates the changes to each hashIn table in order to accommodate an entry for each pair of corresponding places found similar.

If V_3 is further processed by Algorithm 14, a new meta-group

$$M_3=\{\texttt{P11}\leftarrow\texttt{T8}\ \leftarrow\texttt{P7}^{\leftarrow T6}_{\leftarrow T5},\ \texttt{P10}\leftarrow\texttt{T9}\leftarrow\texttt{P8}^{\leftarrow T7}_{\leftarrow T4}\}$$

is formed out of each subnet S$\in V_3$ as illustrated in Figure 5.19. For each place P$\in$ S.iList, indices of its input transitions are found and inserted into hashInIndex[T] as shown in Table 5.10. Subsequently the subnets in M_3 are compared to establish new groups. For any two components E, F$\in V_2$ that were formed out of places pl_i, pl_j $\in V_1$: pl_i appears before pl_j in V_1, the similarity of E and F can be determined by obtaining hash-table $hashIn_{ij}$(=pl_i.inCompare[pl_j]) and checking if hashInIndex[v1]=hashInIndex[v2] for each entry (v1,v2) in hashIn. These hash-tables are shown in Figure 5.19. Comparing the sets returned by hashInIndex for each pair of vertices in hash-table hashIn,

$$\text{hashInIndex[P8]} \sim \text{hashInIndex[P7]} \sim \{2, 1\}$$

the subnets in M_3 are found to be identical. Consequently the new group V_4 is same as M_3.

Table 5.11: Initialisation of hashInIndex is followed by a change in iList for each subnet in M_4

Initial inList	P10.inList={T4, T7}
hashInIndex Initialisation	hashInIndex[T4]={hashFun(P6)}={4}, hashInIndex[T7]={hashFun(P9)}={3}
Updated inList	P10.iList={P6, P9}
Initial inList	P11.inList={T5, T6}
hashInIndex Initialisation	hashInIndex[T5]={hashFun(P4)}={6}, hashInIndex[T6]={hashFun(P5)}={3}
Updated inList	P11.iList={P4, P5}

$$V_4{=}M_3{=}\{\texttt{P11}{\leftarrow}\texttt{T8}\ {\leftarrow}\texttt{P7}^{\leftarrow T6}_{\leftarrow T5},\ \texttt{P10}{\leftarrow}\texttt{T9}{\leftarrow}\texttt{P8}^{\leftarrow T7}_{\leftarrow T4}\}$$

The new group illustrated in Figure 5.20 using red-borders. The figure also shows hash-table hashIn_{10-11} containing two entries (T7,T6) and (T4,T5) that were found to be similar.

Finally, on applying Algorithm 14 again on V_4, a meta-group M_4 is obtained as shown in Figure 5.20.

$$M_4{=}\{\texttt{P11}{\leftarrow}\texttt{T8}\ {\leftarrow}\texttt{P7}^{\leftarrow T6\leftarrow P5}_{\leftarrow T5\leftarrow P4},\ \texttt{P10}{\leftarrow}\texttt{T9}{\leftarrow}\texttt{P8}^{\leftarrow T7\leftarrow P9}_{\leftarrow T4\leftarrow P6}\}$$

The hash-table hashInIndex is updated as shown in Table 5.11. Subsequently the subnets in M_4 are compared to establish new groups. On comparing the sets returned by hashInIndex for each pair of vertices in hash-table hashIn,

$$\text{hashInIndex[T7]}{\sim}\text{hashInIndex[T6]}$$
$$\text{and}$$
$$\text{hashInIndex[T4]}{\not\sim}\text{hashInIndex[T5]}$$

the input place of T7 and T6 are found to be similar while those of T4 and T5 are found to be distinct. Consequently the input places of T4 and T5 are removed to obtain new group V_5 as shown in Figure 5.21.

$$V_5{=}\{\texttt{P11}{\leftarrow}\texttt{T8}\ {\leftarrow}\texttt{P7}^{\leftarrow T6\leftarrow P5}_{\leftarrow T5},\ \texttt{P10}{\leftarrow}\texttt{T9}{\leftarrow}\texttt{P8}^{\leftarrow T7\leftarrow P9}_{\leftarrow T4}\}$$

Furthermore, the hash-table now contains a single entry (P9,P5) as the other pair of places were found to be dissimilar.

The subnets identified in example net is shown in Figure 5.22. Any further attempts in including more places or transitions into these subnets would result in overlapping and consequently these components would no longer constitute a group.

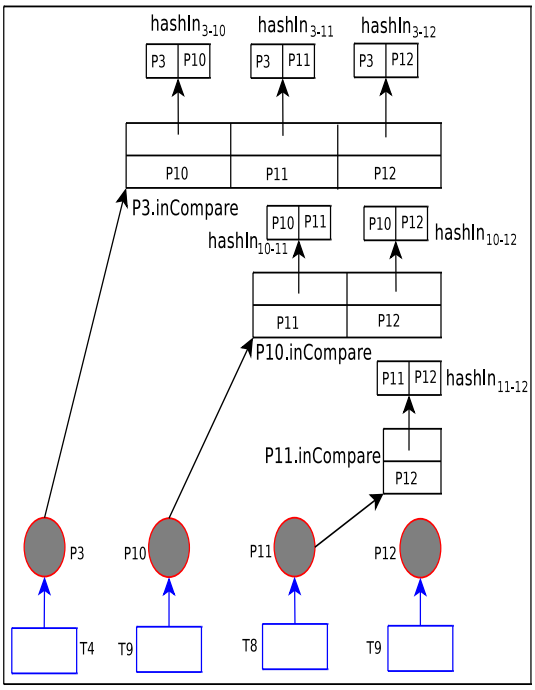


Figure 5.17: The hash-tables in meta-group M_1

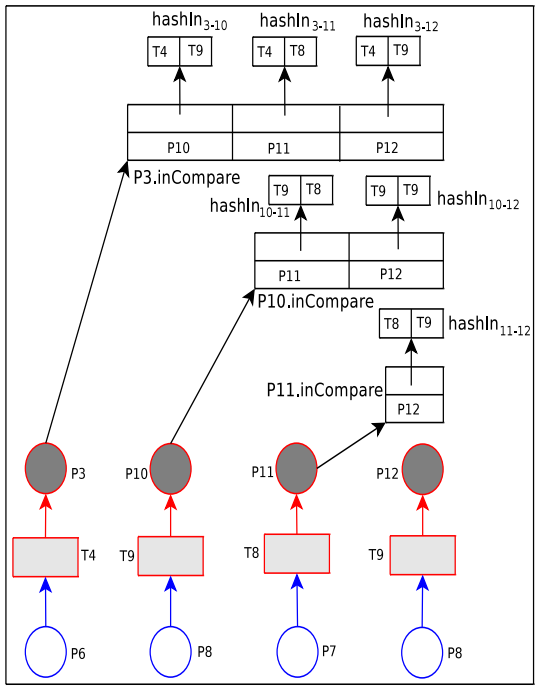


Figure 5.18: The hash-tables in meta-group M_2

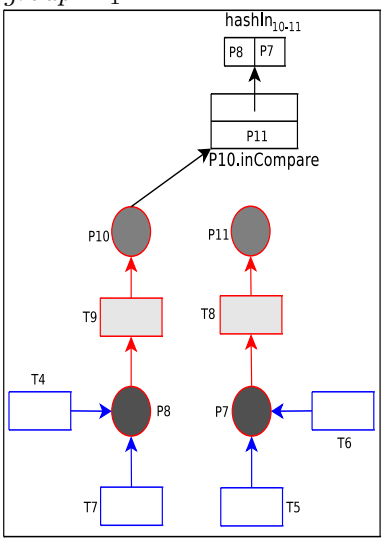


Figure 5.19: The hash-tables in meta-group M_3

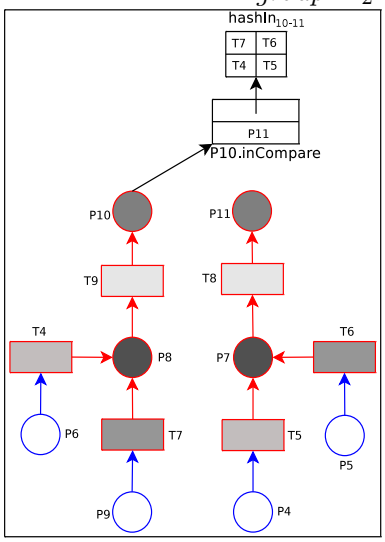


Figure 5.20: The hash-tables in meta-group M_4

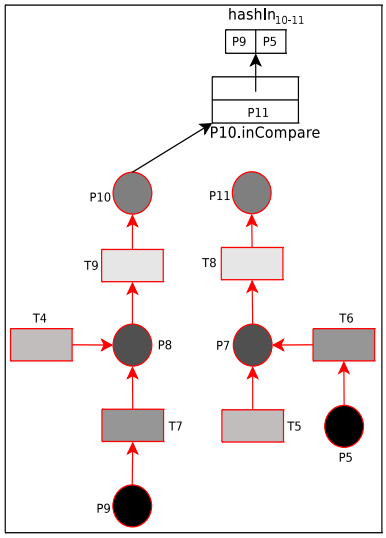


Figure 5.21: The subnets in group V_5

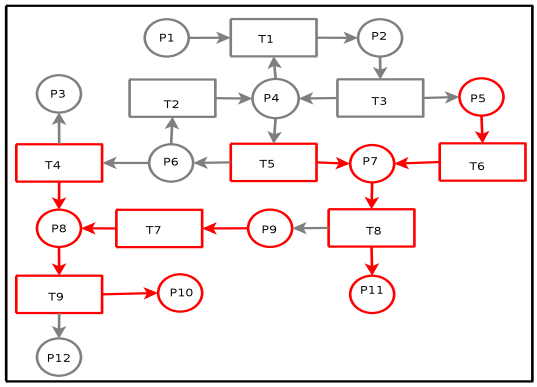


Figure 5.22: The subnets identified in CPN model.

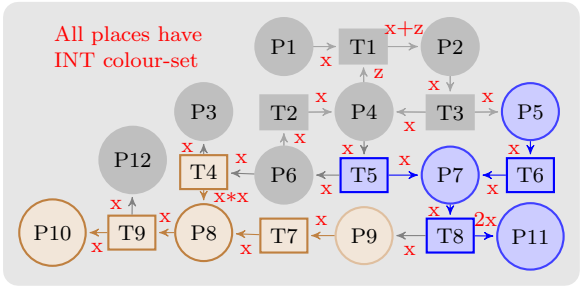


Figure 5.23: The two identical components (blue and brown) forming a group in example net

5.5.2 The Clustering Method

The method proposed herein allows establishing hierarchy into a CPN model by creating a module out of each non-overlapping group identified by the Lookup method. Similar methods can be proposed to install hierarchy into a non-CPN model by using the appropriate semantics for hierarchy as defined. The steps involved are explained herein with the example net in Figure 5.23 and its hierarchical counterpart in Figure 5.24.

1) Creating new modules: A new module is created for each non-overlapping group based on the structure of components in the group. If a transition in the new module has fewer adjoining places than it has in the original net, supplementary places are created and attached to it. This ensures that each socket in supermodule has a port in submodule. Figure 5.24 illustrates the new module *g2* created from components identified in Figure 5.23. Additional ports are attached to E1 and E2 (dotted places) corresponding to the adjoining places for T4/T5 and T9/T8.

2) Declare Union colour-sets if necessary: On superimposing the components in a

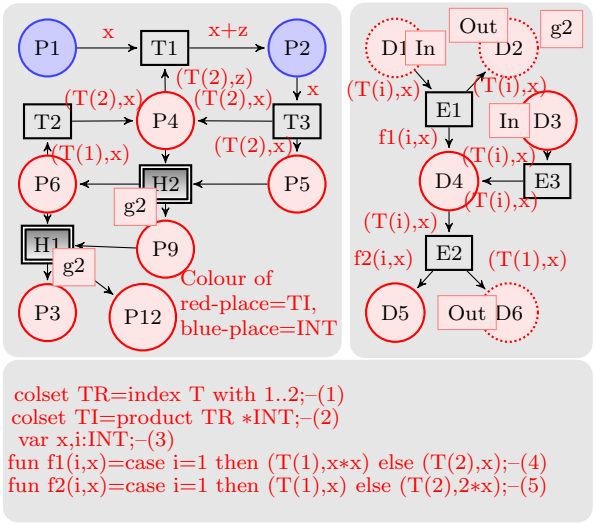


Figure 5.24: The equivalent hierarchical model obtained using the Clustering algorithm

group, the places and transitions in each component would overlap with their counterpart in other components. However, if the colour-sets for any bunch of overlapping places are not the same, a new colour set needs to be defined as the union of all these colour sets and assigned to their counterpart in the new module. This ensures that their corresponding place in the new module can hold a token from any (or all) of them. For instance if P7 had STRING colour-set in Figure 5.23, it would be in conflict with the INT colour set of P8. Accordingly a union INTnSTR needed to be declared as

colset INTnSTR = union Int:INT+String:STRING;

and this would be assigned to corresponding place D4 Figure 5.24 (after declaring product colour-set, step 5). Furthermore if any port-socket pair has disparate colour-sets, the union of their colour-set is assigned to the port and all its associated sockets. This ensures that port-socket pairs have consistent colour-sets.

3) Delete places and transitions from original net: For each component in a group, its constituent places and transitions are deleted from the original net with the exception of peripheral places that constitute the sockets in supermodule. The deleted nodes are subsequently replaced by substitution transitions with a tag containing the name of the new module. Any arc from socket places to a deleted node is now connected to substitution transition. Figure 5.24 illustrates the nodes deleted from Figure 5.23.

4) Assign ports: The ports are tagged (*In*, *out* or *I/0*) and mapped to their counterpart

(sockets) in original net.

5) Declare index colour-sets: For each group, an index colour set is declared of the same size as the group. The elements in this colour set are used to identify the components that were replaced by substitution transition. This allows us to introduce component specific behaviour in the new module. For instance the two components of example net differ in a few arc-expressions. To account for this difference, an index colour set has been defined as equation 1 of Figure 5.24. Furthermore, the expression on arcs with a socket at either end is modified to include the identity of component consuming the token (e.g. input arcs of P4,P5). Accordingly the colour of socket places and the places in new module are also modified to contain component information (colour TI for red-places).

6) Declaring the functions: The component specific behaviour of transitions in a component is obtained by declaring additional functions. For instance function f2 doubles the value in token (2∗x) for second component (i=2) while it keeps the value same for first component (i=1). Such component specific behaviour of transitions ensure that the behaviour of original net is not changed on installing hierarchy.

7) Tokens on output ports: The place P6 in Figure 5.24 is the output socket for H2 and the input socket for H1. Consequently before adding tokens to P6, H2 updates the component information in token (input arc-expression of D6). Failing to do so will cause H1 to emulate the behaviour of H2 and this might lead to potential errors.

The hierarchical-model returned by the Clustering method is shown in Figure 5.23.

5.5.3 Time Complexity of the Lookup algorithm

Consider a net N with n_p places and n_t transitions such that
(i) Maximum indegree of places(Δ_p^-)=m_p
(ii) Maximum outdegree of places(Δ_p^+)=l_p
(iii) Maximum indegree of transitions(Δ_t^-)=m_t
(iv) Maximum outdegree of transitions(Δ_t^+)=l_t.
Hence $\forall i \in [1,n_p]$, $deg^-(p_i) \in \{0, 1, 2, \cdots, m_p\}$ and $deg^+(p_i) \in \{0, 1, 2, \cdots, l_p\}$, where deg^- and deg^+ denote the indegree and outdegree of any node. Considering that a place in net N can have (m_p+1) possible values for deg^- and (l_p+1) possible values for deg^+, the number of rows in hash-table for places would be

$$x_p \leq ((m_p + 1) * (l_p + 1) - 1) \tag{5.10}$$

This value is 1 less than the product because both $\deg^-$ and $\deg^+$ cannot be 0 for a place. Similarly, the number of rows in hash-table for transition would be

$$x_t \leq ((m_t + 1) * (l_t + 1) - 1). \tag{5.11}$$

If the probability that a place is at index k($1 \leqslant k \leqslant x_p$) is $Prob_p(k)$, the number of places at index k of hash table would be n_p*Prob(k). Consider functions In_p:[1,x_p]$\rightarrow$[0,m_p] and Out_p:[1,x_p]$\rightarrow$[0,l_p] that return the $\deg^-$ and $\deg^+$ of places at some index k($1 \leqslant k \leqslant x_p$). If z vertices could be compared in time T(z), the time taken to compare input transition of a place at k would be T(In(k)). Similarly, time for comparing output transitions of a place at k would be T(Out(k)). Consequently, the comparison time for all n_p*Prob(k) places at index k would be

$$n_p * Prob(k) * (T(In(k)) + T(Out(k))). \tag{5.12}$$

The overall delay in comparing places at each index of hash table would be

$$\sum_{k=1}^{x_p} n_p * Prob(k) * (T(In(k)) + T(Out(k))) \tag{5.13}$$

$$or \quad n_p \sum_{k=1}^{x_p} Prob(k) * (T(In(k)) + T(Out(k))) \tag{5.14}$$

Equation (7) implies that the delay is linear on n_p. Similarly, it can be certified that the delay is linear on n_t. Creating the hash-tables is another linear process wherein the entire set of vertices are scanned once in each of the two passes. In the first pass, the number of times inner `foreach` loops execute for a vertex v_i is $\deg^-(v_i)$+$\deg^+(v_i)$. However, these values are independent of n_p and n_t. Evaluating the hash-function in second pass is also a constant time operation. Consequently, the time complexity of the Lookup algorithm is $\theta(n_p+n_t)$.

5.6 Experimental Results

The proposed algorithm was implemented using Qt 4.5 SDK for 32-bit linux. The algorithm was tested on a PC with 1.83GHz Intel Core 2 Duo processor and 2GB of RAM. The PC had Ubuntu 8.04 desktop version OS installed and our C++ code was compiled using GNU g++ compiler. The implementation could be downloaded at [Mukherjee, 2009b].

A net consisting of all dissimilar places and transitions would have a single entry at each index of the hash-tables generated by Algorithm 12. When these elementary groups are fetched by Algorithm 13 and sent to Algorithm 14, the latter quits at the very first step. This is because Algorithm 14 requires groups with multiple subnets for processing. Consequently, the time taken by the Lookup algorithm would be minimum. On the contrary if there are large number of identical components in a net, a group passed to Algorithm 14 would have many subnets. Consequently, Algorithm 14 might need to execute several times recursively before the size of group reduces to one and causes the algorithm to stop. Therefore given a net N, the best case occurs when it has no identical components and the performance of algorithm degrade with introduction of identical components into it.

In order to find the effect of incorporating identical components on the run-time of the Lookup algorithm, the net in Figure 5.7 is used to create larger nets by joining multiple instances of it in various patterns. Table 5.12 illustrates the patterns in which this net is joined to create new nets. Each bullet (•) represent the net in Figure 5.7. The table assigns a number to each of these nets. The structure of net 17, which is the supergraph of all other nets, could be found at [Mukherjee, 2009a]. The structure of other nets could be interpreted from this structure. In Figures 5.25, 5.26, 5.27 and 5.28, the time taken by the Lookup algorithm to process these nets is exhibited in red. However, modifying the structure of these nets to reduce the number of structurally similar components lead to a reduction in execution time of the Lookup algorithm. The reduced execution times are shown in blue. The possibility of being able to reduce the execution time by modifying the net structure indicate that the Lookup algorithm depend on the number of structurally similar components in a net, apart from its size. This is later demonstrated in further detail using Figures 5.29, 5.30, 5.31 and 5.32. The experiment is run 50 times for each net and the average execution time is recorded as the time of execution for that particular net.

Figure 5.25 illustrate the time taken by the Lookup algorithm to find all groups in nets 1-7. The processing time increase as new instances of the aforesaid net are attached. The time taken by the Lookup algorithm to process nets 2 and 3 differ marginally as they are composed of same number of instances of elementary net in Figure 5.7. This is also true for nets 4, 5 and 6. However, a net with two instances of elementary net joined vertically (net 2) has slightly greater processing time than net 3, wherein the instances are joined horizontally. This indicates the presence of additional identical components in net 3. For similar reasons, net 5 takes longer to process when compared to net 6 .

Figure 5.26 illustrate the time taken by the Lookup algorithm to process nets 8-12. The

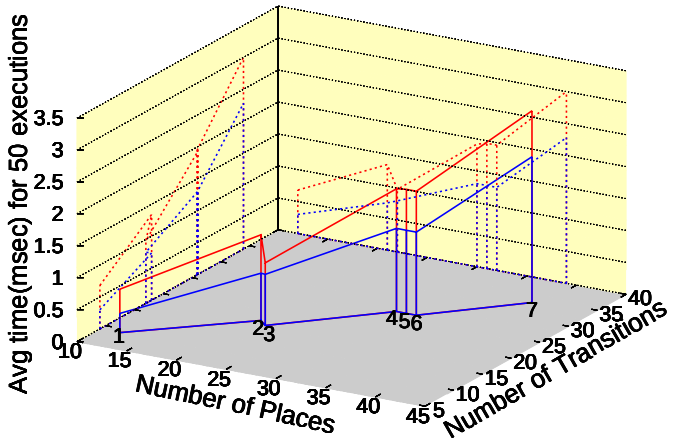


Figure 5.25: Average time (in msec) taken by the Lookup algorithm for nets 1-7. The dotted-lines indicate the projection of curve on either axes.

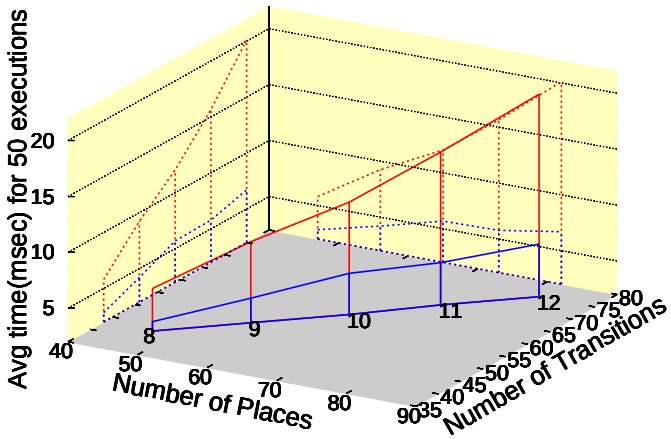


Figure 5.26: Average time (in msec) taken by the Lookup algorithm for nets 8-12. The dotted-lines indicate the projection of curve on either axes.

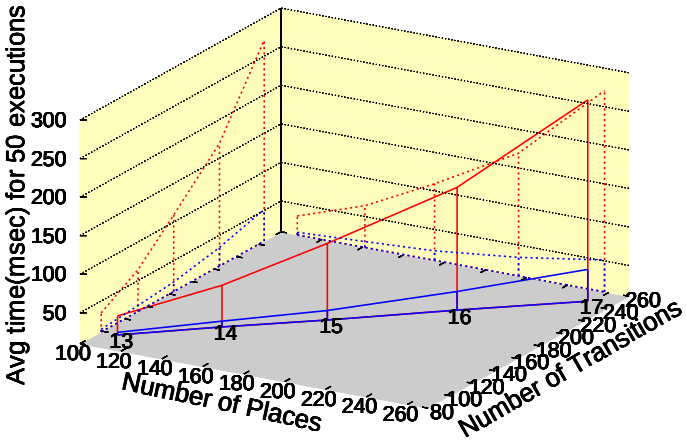


Figure 5.27: Average time (in msec) taken by the Lookup algorithm for nets 13-17. The dotted-lines indicate the projection of curve on either axes.

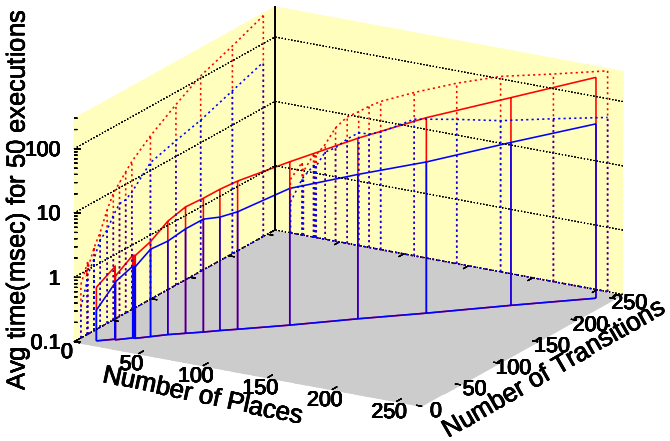


Figure 5.28: Average time (in msec) taken by the Lookup algorithm for each net (1-17). The dotted-lines indicate the projection of curve on either axes.

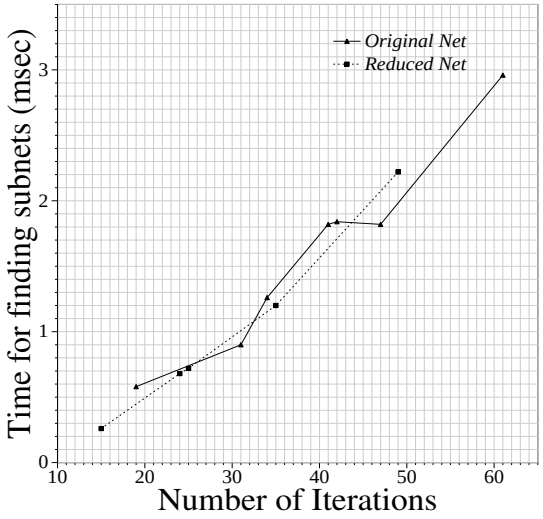


Figure 5.29: Average time (in msec) vs Number of times Algorithm 14 is invoked for nets 1-7.

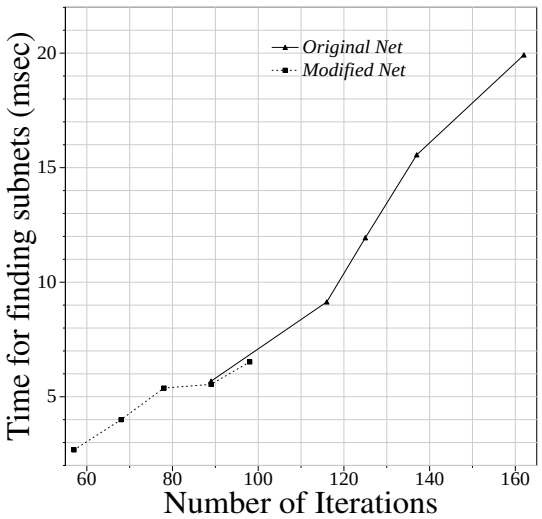


Figure 5.30: Average time (in msec) vs Number of times Algorithm 14 is invoked for nets 8-12.

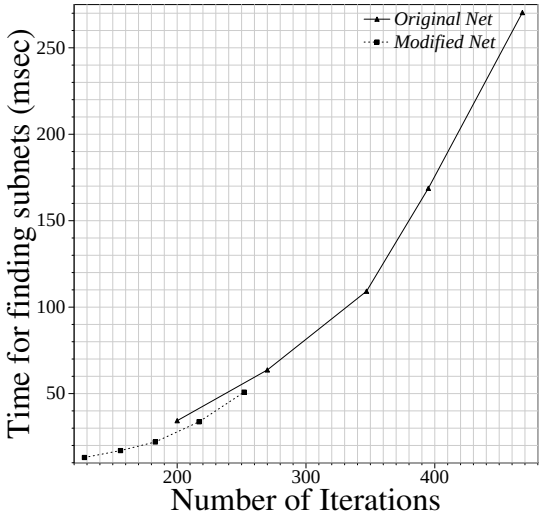


Figure 5.31: Average time (in msec) vs Number of times Algorithm 14 is invoked for nets 13-17.

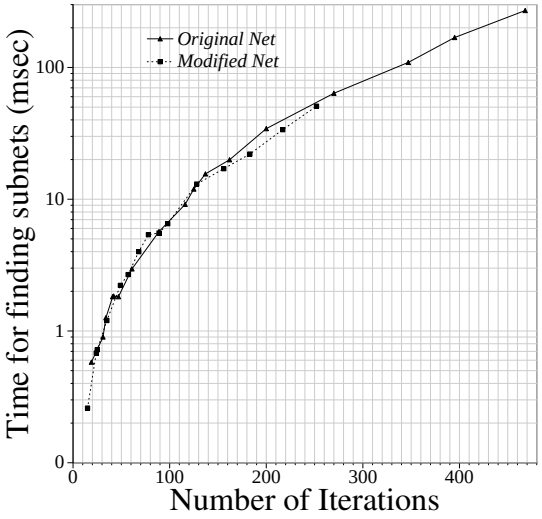


Figure 5.32: Average time (in msec) vs Number of times Algorithm 14 is invoked for all nets. Note that the curves overlap.

Table 5.12: The nets [Mukherjee, 2009a] used for analysing execution time of the Lookup method

#	Net	#	Net	#	Net	#	Net	#	Net	#	Net	#	Net
1		2		3		4		5		6		7	
8		9		10		11		12		13		14	
15		16		17									

increase in processing time is by and large linear as new instances of the elementary net are added. The processing time for nets 13-17, exhibited in Figure 5.27, increase more rapidly when compared to smaller nets considered hitherto. However, the deviation from linear path for the curve is not large. Figure 5.28 compare the execution time for all 17 nets together. The increase in execution time is linear for the first 14 nets. Subsequently, the execution time increase more rapidly. However, considering that net 17 has 245 transitions and 258 places, an execution time of 270 msec can be considered fairly reasonable.

Furthermore, apart from the size of a net, the processing time also depends on number of structurally similar components in it. This is demonstrated in Figures 5.29, 5.30, 5.31 and 5.32 where the execution time for Algorithm 14 is plotted against its number of invocations. As discussed previously, the algorithm will be invoked more frequently in case of a net with many similar components. Comparing the curves corresponding to the original and modified nets, the processing time for an original net is found to be equal to that of a modified net of much smaller size provided they invoke the algorithm same number of times. The latter is an indication of their structural similarity. Considering the overlap between the curves, it can be deduced that 'as the number of similar components in a net is reduced, its processing time will skid down the curve'. This implies that the processing time for a net depend on the number of identical components in it.

Figures 5.33 and 5.34 demonstrate the time taken by Algorithms 14 and 12 for each net. Considering that the latter takes a minuscule time for processing, the execution time of the Lookup algorithm is in effect the time taken by the former.

We have considered the worst case scenario in Figures 5.29, 5.30, 5.31 and 5.32 wherein the number of identical components have been maximised by using multiple instances of the same elementary net in creating larger nets. As demonstrated previously, the execution time depend on number of identical components in a net apart from the size of net. Consequently, reducing the number of identical components in a net is followed by reduction in execution

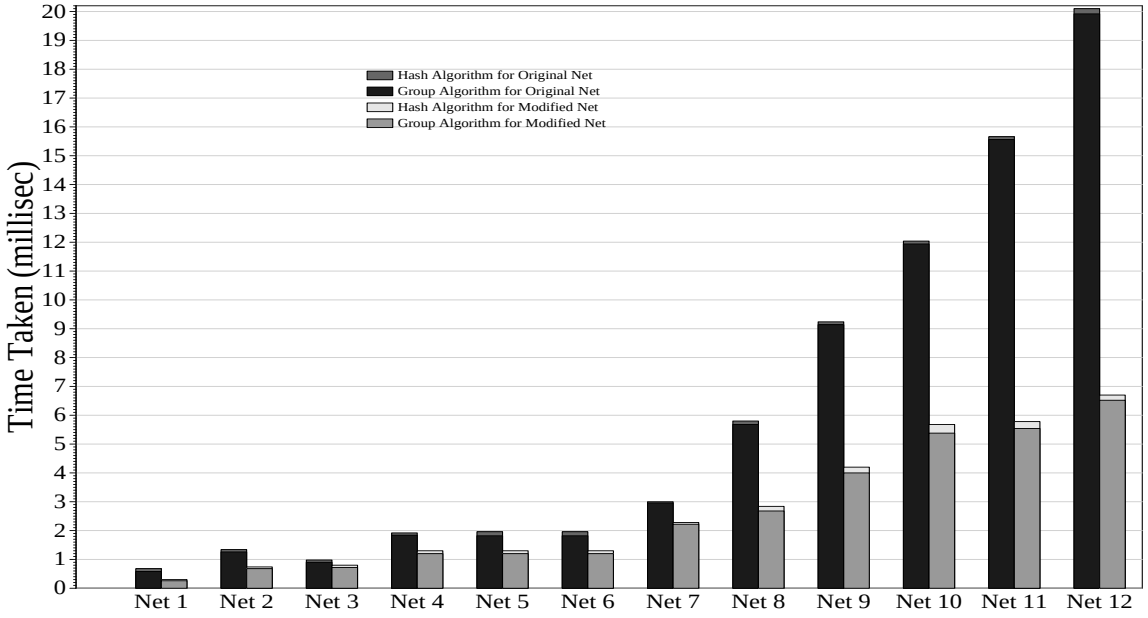


Figure 5.33: Processing time of Algorithm 14 and Algorithm 12 for nets 1-12.

time.

5.7 Discussion

This chapter identifies the problems in analysing flat models and proposes a novel solution to address them. The complex and concurrent components in a contemporary software system often leads to a massive increase in size of the corresponding formal representation. This is further exacerbated by the lack of abstraction and hierarchy in this representation. Consequently a human modeler is required to deal with an overwhelming stockpile of information while analysing a flat model. Such state of affairs could lead to potential disasters owing to any errors and omissions. We propose to prevent such scenarios by introducing hierarchy into an otherwise flat model. This allows a human modeler to analyse and acquire a better understanding of the system owing to the introduced modularity and abstraction.

The proposed solution is outlined in Figure 5.5. The envisioned hierarchy is installed by identifying the structurally similar components in a flat model and creating a module for each one of them. The Lookup method identifies the identical components bottom-up starting from the elementary components of a model. It is based on the decrease and conquer strategy wherein the solutions to the smaller problems are used to solve a bigger problem. The results indicate that the proposed Lookup method takes linear time to find the groups

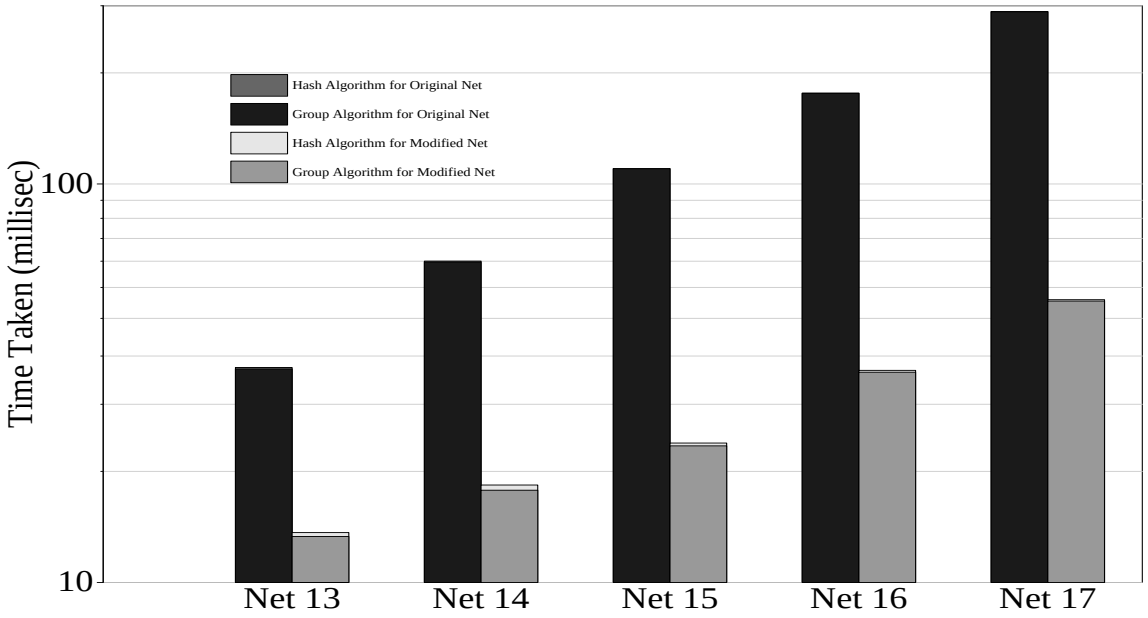


Figure 5.34: Processing time of Algorithm 14 and Algorithm 12 for nets 13-17.

in a sufficiently large net. Furthermore, the execution time of the Lookup method is found to be dependent on the number of identical components in a model.

As discussed previously, the best case for the Lookup method necessitates the absence of any identical components. Such a scenario compels Algorithm 14 to quit without any processing. However, as reflected in Figures 5.29 to 5.32, an increase in the number of identical components escalate the iterations of this algorithm and the associated delay.

In order to study the effect of adding additional components on delay, multiple instances of the example net are joined as shown in Table 5.12 (• represents the example net in Figure 5.7). From Figure 5.25, the time for processing the example net is .68 msec. When this net is attached to another instance of itself, the processing time is found to increase by more than twice. This is because in addition to the size, the number of identical components also double. Each component (places and transitions) in the first instance would have a counterpart in the other instance leading to elementary groups of twice the size. Considering that 1) each iteration of Algorithm 14 attaches only a single layer of peripheral nodes to the components in a group, and 2) the size of identical components could be as large as the example net itself (because the model was formed by attaching two identical instances of example net), the algorithm needs large number of additional recursions to determine all identical components. The required recursions will further increase as additional instances

of identical components are added. Consequently the execution time in Figures 5.25 to 5.28 increase with an increase in number of identical components.

The overlapping of curves for original and modified nets in Figures 5.29 to 5.32 further endorse the dependency of execution time on the number of identical components. When a net is modified to ensure it has fewer identical components, its execution time is found to skid down along the curve. Consequently the execution time of a larger net (e.g. modified net 17 in Figure 5.31) is found to be less than a much smaller net (original net 14 in same figure) when the former is modified to reduce the number of identical components. The consistency of this trend confirms the aforementioned dependency.

Figure 5.28 compares the average execution time for all 17 nets considered. Although the execution times are fairly linear for the first 14 nets, they increase more rapidly thereafter. This increase can be attributed to the comparison of indices in the second phase of the Lookup method. Each entry in the hash-tables *hashInIndex* and *hashOutIndex* consists of a list of indices corresponding to a vertex. As mentioned previously, these lists contain the indices for the input or output nodes corresponding to the key vertex. The delay is attributed to the comparison of these lists in step 32 of Algorithm 14. For big nets, these lists get significantly large resulting in the observed delay.

5.8 Summary

In this chapter, we have proposed the Lookup and the Clustering algorithms to install hierarchy into a CPN model. While the former identifies structurally similar components in a net, the latter uses these components for installing hierarchy. The Lookup algorithm is based on 'Decrease-and-Conquer' strategy wherein the bigger problem is broken into smaller problems and the solution to smaller problems are combined to solve the original problem. It is a generic algorithm and can be used for a wide array of modeling languages that define a notion of hierarchy and structural similarity. On the contrary, the Clustering algorithm uses the semantics of hierarchy defined for a particular modeling language and is consequently specific to the language. The results indicate that even in the worst case scenario, the Lookup algorithm has linear time complexity for sufficiently large nets.

The proposed solutions addressing a niche for contemporary software systems that have a gigantic formal representation owing to the high levels of concurrency and complexity. Our technique allows a human modeler to comprehend, analyse and maintain the representation by exploiting the abstraction and modularity. Considering that formal methods are applied

at the early stage of application development, a better understanding at this phase would significantly enhance the quality and reliability of the final product.

Chapter 6

A Framework for Modeling, Simulation and Verification of a BPEL Specification

The previous chapters have described methods for reducing the time and memory requirements for model-checking. These methods offer incentives in using formal methods for verifying Service-Oriented architecture (SOA) based applications. These chapters have have also emphasised the ingenuity of Formal-Methods (FMs) in endorsing the safety and reliability of software-systems. After their widespread use in software engineering, they are being increasingly adopted for verification of SOA based applications [Foster et al., 2003; Kang et al., 2007; Yang et al., 2005; Yi and Kochut, 2004]. This is helping in identifying subtle errors in a *Business Process Execution Language (BPEL)* specification, the de-facto industry standard for service composition. Such errors would often elude conventional simulation and testing techniques.

SOA based applications have assumed widespread acceptance owing to their agility, maintainability and modularity. However, the safety and reliability of such loosely coupled systems entirely depend on the precision of service descriptions. Consequently any implicit assumption or unforeseen usage scenarios can lead to catastrophic fiascos. This is further exacerbated by the overlapping constructs and inconsistencies in BPEL.

Conventional techniques cannot be applied for verifying a SOA based application because 1) the fault, if any, is mostly related to the business logic for service-composition rather than the source-code or implementation of underlying services; 2) even if an issue is found with

the implementation of a service, the source code is usually not available for rectification; and 3) even if the source code is available, it cannot be immediately rectified as this might probably break thousands of other applications using this service. Consequently an appropriate verification technique would investigate the service composition for all possible behaviours of the underlying services.

In recent years, several sophisticated techniques have been proposed to allow automatic matching and discovery of web-services [Hao and Zhang, 2007; Paolucci et al., 2002]. This empowers automatic and dynamic location of suitable web-services and their composition to offer an envisioned complex service. Nevertheless, such techniques would overwhelmingly rely on automated verification methods to vouch for their credibility. Considering that model-checking is an automated verification technique that scrutinises all possible behaviours of a system exhaustively, it ought to be used for SOA based applications. However, in the absence of a concrete framework, this verification process is essentially ad-hoc.

This chapter extends the *Spring framework* to devise a verification framework for service composition wherein each BPEL activity is represented by a *Java bean*. This framework instantiates the beans corresponding to activities in a BPEL specification and injects the dependencies to yield a *bean-factory*. Thereafter *Java Architecture for XML Binding (JAXB) 2* APIs are used to transform the bean-factory into an XML based formal-model (e.g. Coloured Petri nets (CPN)) or an interchange format (e.g. Petri Net Markup Language (PNML)) for simulation and verification. In addition to automating the verification process, the proposed framework helps to combat the ad-hoc nature of existing solutions. Results indicate that the framework takes .7 sec on an average for formalisation of a BPEL specification.

To evaluate the framework, a CPN template (which is XML based) is proposed for each BPEL activity. The JAXB 2 APIs generate the formal model based on these templates. Each template is customized based on the attributes specified for the corresponding BPEL activity. A CPN model is essentially an XML document and these templates confirm to the Document Type Definition (DTD) supplied for CPN tools [Westergaard et al., 2005].

6.1 Motivation

SOA based applications are built as an assembly of existing web-services that are invoked in some sequence based on the underlying business logic. Its component web-services can span across several organizational boundaries and have any underlying implementation. Such state of affairs have necessitated a tool for orchestrating the business workflow. Among all

the domain-specific languages that were proposed, the Business Process Execution Language for web-services [Curbera et al., 2002; Andrews et al., 2003; Arkin et al., 2005] stands out as the de-facto industry standard for *web-service composition*.

SOA based applications offer code mobility, cross client support, code reuse, better scalability and distinct partition of application layers. However, the safety and reliability of such loosely coupled systems entirely depend on the precision of service descriptions. Consequently any implicit assumption or unforeseen usage scenarios can lead to undesirable forms of interactions, such as a deadlock or race condition [Sloan and Khoshgoftaar, 2009].

Considering the unprecedented ability of formal-methods in ensuring the correctness of a system, they ought to be adopted for verifying SOA based applications [Clarke et al., 2000; Merz, 2001]. This would help in identifying subtle errors in BPEL specification that often elude conventional simulation and testing techniques.

Unfortunately the overlapping constructs [Wohed et al., 2002] and the lack of sound formal or mathematical semantics [van der Aalst, 2003; Schmidt and Stahl, 2004] in BPEL does not allow formal methods to be applied into its textual specification. These inconsistencies are the outcome of two conceptually contrasting languages (Web Services Flow Language (WSFL) [IBM, 2001] of IBM and XLANG [Thatte, 2001] of Microsoft) that were amalgamated to constitute BPEL [Schmidt and Stahl, 2004].

Consequently the textual specification of BPEL needs to be transformed into a formal specification prior to any formal verification. A BPEL specification is essentially a sequence of activities and the aforesaid transformation involve formalization of each of these activities.

Most of the existing solutions incorporate an ad-hoc mapping of BPEL activities into formal-models [Foster et al., 2003; Kang et al., 2007; Yang et al., 2005; Yi and Kochut, 2004]. Users are required to scan a BPEL specification and replace each activity with its corresponding formal-model. Apart from being a cumbersome process, such an exercise is error-prone and time-consuming. Although there exist some solutions that automate this translation, they are neither generic nor provide pluggable interface to qualify as a framework [Fu et al., 2004]. Furthermore, they do not consider BPEL's most interesting and complicated activities like *eventHandler* and *links* and overlook crucial scenarios such as *Dead-Path elimination*(DPE).

The crux of the problem lies in attempting to formalise a BPEL specification using a specific modeling language. Considering that there are many modeling languages available (e.g. Promela, Petri Nets, Automata, Process Algebras etc.), targeting a specific language renders an ad hoc and temporary solution. Consequently this chapters transforms a BPEL

specification into a generic intermediate specification formed before the actual formalisation. In software engineering, Data Transfer Objects (DTOs) are commonly used design patterns for storing and transferring data [Crawford and Kaplan, 2003]. Therefore we use DTOs to store the generic intermediate specification, wherein each BPEL activity is mapped to a separate DTO.

The aforementioned intermediate specification is obtained from a BPEL specification using Spring framework. Despite the syntactical differences between a BPEL specification and a Spring configuration document, this chapter identifies immense semantic similarity. Consequently Spring framework has been extended to recognise BPEL activities and populate corresponding DTOs. This helps in significantly automating the envisioned transformation.

As observed previously, the generic intermediate specification could be transformed into any modeling language. However, this chapter specifically targets the modeling languages that are based on XML. This is done using the JAXB 2 APIs that transform the Spring bean-factory into an XML based formal-model or an interchange format. The latter acts as an intermediate specification that can be transformed into a range of formal models. For instance PNML [Billington et al., 2003] can yield different versions of Petri-Nets.

Our contributions can be summarised as:

1. We propose a Spring based verification framework to map individual BPEL activities into Java beans. This offers several advantages over existing techniques that map BPEL activities into formal models: a) the transformation is automatic b) Java beans are generic intermediate specifications that can yield a range of formal models c) the framework significantly reduces the time required for transformation.

2. We introduce an object model to streamline the mapping of BPEL activities into Java beans. This object model identifies the relationship among BPEL activities and forms the basis of 1) mapping from BPEL activities into DTOs and 2) CPN templates for BPEL activities.

3. We propose a JAXB 2 based component for the framework to transform the bean-factory into an XML based formal model. It forms a non-core component that could be replaced to transform the bean-factory into non-XML formal models. The pluggability of formalising component into the proposed framework forms the basis of its flexibility.

4. We have formalized BPEL activities as XML templates[1] that confirms to the DTD

[1] templates are part of a complete formal model

specified for CPN Tools. These templates are used by JAXB 2 compiler to generate the formal models. The templates exploit the hierarchical relationship among BPEL activities that is defined using the object model.

The remainder of the chapter is organized as follows. Section 6.2 introduces the deliberated problem and provides an insight into the tendered solution. Prior to proposing the templates for formalizing a BPEL specification in Section 6.4, the existing works are compiled in Section 6.3. The experimental results are presented in Section 6.5 and the outcome is discussed in Section 6.6. Finally we summarize our contributions in Section 6.7.

6.2 An Overview of the Deliberated Problem & the Tendered Solution

In this section, we discuss the problem in detail and outline the proposed solution. As pointed out previously, BPEL has emerged out of conflicting parent languages that vastly differ in the structure of control and message flows. Consequently it is infested with umpteen inherent inconsistencies and ambiguities that seriously undermine the reliability of a SOA based application. This is further aggravated by the loosely coupled nature of a SOA based application. The safety and reliability of such loosely coupled systems entirely depend on the precision of service descriptions wherein any implicit assumption or unforeseen usage scenarios can lead to undesirable forms of interactions, such as a deadlock or race condition [Sloan and Khoshgoftaar, 2009]. Furthermore, as observed previously, dynamic service composition techniques overwhelmingly rely on automatic verification techniques to vouch for their reliability.

As emphasized in previous chapters, *model-checking* [Clarke et al., 2000] is an automatic formal-verification technique that is being increasingly adopted as a standard procedure for quality assurance of software systems. Taking cue from this tenacious trend, this chapter proposes a verification framework to enhance the reliability and correctness of SOA based applications. This has been done by mapping each BPEL activity into a Java-bean that is instantiated and initialised by Spring framework. This mapping is essentially between the properties of a class and the attributes of its corresponding BPEL activity. Furthermore, the classes corresponding to BPEL structured activities have an additional property to store the list of its child activities. The Spring framework has been extended to accept a BPEL configuration and to instantiate and initialise the classes (using their setter methods) corresponding to the activities in the specification. The objects (java-beans) rendered act as intermediate specifications that could be transformed into a range of formal models. For the purpose of verification in this chapter, a pluggable component is proposed that uses JAXB

2 APIs to render XML based models.

In order to generate a formal representation, the pluggable component requires an XML based model corresponding to each BPEL activity. Considering that CPN model are XML based, we have proposed a CPN template for each BPEL activity. We have established a hierarchical relationship among BPEL activities that allow us to reuse a parent template for its child activities after any required customization. This helps in significantly reducing the number of templates required. Each proposed template confirms to the DTD specified for CPN tools.

As compared to the existing techniques, our solution 1) offers a generic intermediate specification that can be transformed into a range of formal models 2) offers an interface to plug in alternative components for transformation.

6.3 Related Work

All solutions for formalizing a BPEL specification involves a transformation into either of 1) Petri Nets / Coloured Petri Nets [Kang et al., 2007; Yang et al., 2005; Sloan and Khoshgoftaar, 2009; Stahl, 2005], 2) Process Algebras [Ferrara, 2004], 3) Abstract State Machine [Fahland, 2005; Fahland et al., 2005] or 4) Automata [Arias-Fisteus et al., 2004; Fu et al., 2004]. They all offer significant strides in the formal-verification of a BPEL specification. However, [Fu et al., 2004; Kang et al., 2007; Yang et al., 2005] do not consider BPEL's most interesting and complicated activities like eventHandler, links and *Dead-Path elimination* (DPE) associated with links. Although [Stahl, 2005] include scenarios involving these activities, the rendered models are bulky and error-prone owing to the plain-vanilla Petri Nets used. The abstract state-machine based solutions are also feature complete. However, they lack adequate tool support for simulation and verification.

In order to support different versions of Petri-Nets, an XML-based interchange format has been proposed and is known as Petri Net Markup Language (PNML) [Billington et al., 2003]. The BPEL2PNML tool [Ouyang et al., 2007] exploit this offering by transforming a BPEL process code into a document conforming to the PNML syntax. Thereafter the formal verification and analysis is performed using WofBPEL [Ouyang et al., 2005] tool that is built using Woflan [Verbeek et al., 2001]. However, due to the lack of a robust visual modeling formalism, the technique fails to render an insight into the obtained Petri net (i.e. PNML). The Platform Independent Petri-net Editor (PIPE) tool used for graphical visualization of PNML models is still in its early stages of development [Bonet et al., 2007].

Table 6.1: A comparitive summary of related works.

Related Work	Tech	LN	ECF	TOOL	VIS
[Ferrara, 2004]	PA	⊗	●	⊗	⊗
[Fahland, 2005]	ASM	●	●	⊗	⊗
[Fu et al., 2004]	AM	◐	◐	●	◐
[Kang et al., 2007]	CPN	⊗	⊗	●	●
[Sloan and Khoshgoftaar, 2009]	CPN	◐	◐	●	●
[Stahl, 2005]	PN	●	●	●	⊗
[Ouyang et al., 2007]	PN	●	●	●	◐
Our Technique	CPN	●	●	●	●

As emphasized previously, PNML is an interchange format and can be transformed into different versions of Petri net. The technique in [Sloan and Khoshgoftaar, 2009] builds on [Ouyang et al., 2007] by transforming the PNML model yield by latter into a CPN model. Apart from being inefficient as compared to a direct conversion, there is always a probability of introducing errors into the intermediate Petri Net model due to 1) the lack of robust visual formalism 2) the size of intermediate model that is often too large for human comprehension. The importance of geometry on the understandability of a visible model has been stressed and thereupon incorporated in our work.

Table 6.1 compares the related work on formalization and verification of BPEL under five different categories. These categories are listed under individual columns and their applicability for any related work is documented using three different symbols. The criteria for constituting these categories are:

- *Tech* denotes the technique used for formalizing the BPEL activities. The techniques and their short-codes are as follows : PA for Process Algebra, ASM for Abstract State Machine, AM for Automata, PN for Petri Nets and CPN for Coloured Petri Nets.

- *LN* denote if links can be used to express synchronization dependency in the target model. While [●] denote 'Yes' and [⊗] denotes 'No', [◐] denotes cases where dead-path elimination is not covered.

- *ECF* denote if Event, Compensation and Fault handlers are covered. While [●] denote 'Yes' and [⊗] denotes 'No', [◐] denotes cases where not all the three handlers are covered.

- *TOOL* denote if verification tool is provided for the rendered model. A [●] denote 'Yes' and [⊗] denotes 'No'.

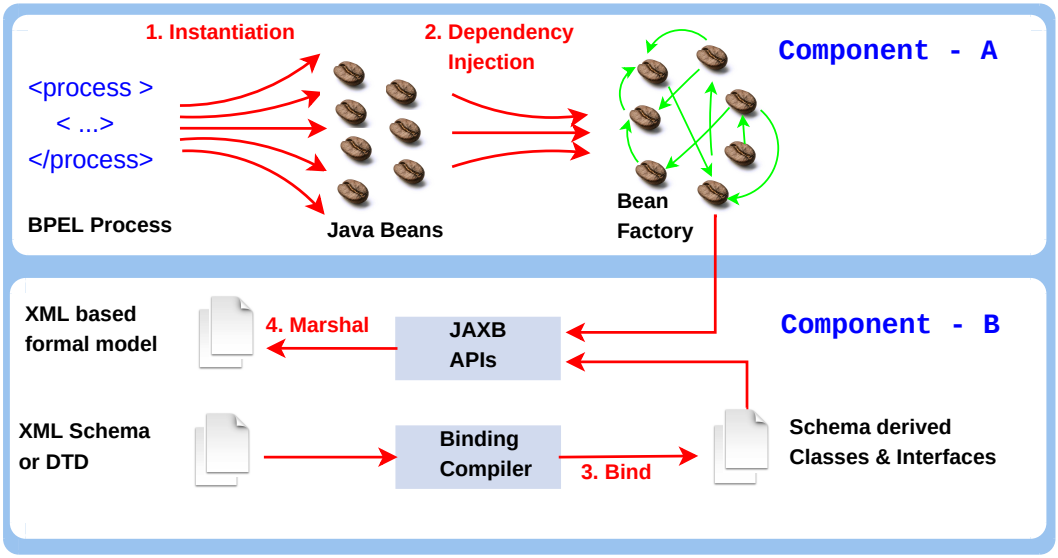


Figure 6.1: The architecture of proposed verification framework.

- *VIS* denote if a robust visualization formalism exists to provide an insight into the transformed model. While [●] denote 'Yes' and [⊗] denotes 'No', [◐] denotes cases where the tool is not robust.

6.4 The Proposed Coloured Petri-Net Semantics for BPEL

This section introduces the framework for automatic formal verification of web-service composition. As illustrated in Figure 6.1, the framework has two basic components. The first component (component - A in Figure 6.1) is an extension of Spring framework that instantiates the beans corresponding to a BPEL specification and renders the bean-factory. It is also the core component of the framework. The other component (component - B in Figure 6.1) transforms the Java beans into an XML based model based on a DTD or an XML schema. This component can be replaced or supplemented with additional components required for transformation.

Figure 6.1 also highlights the four steps that are involved in transformation. While the first two steps involve component - A, the other two steps involve component - B. The role of each of these steps and their prominence in a component are further discussed in the following sections.

```
<partnerLink name="PL1"                <bean id="PL1" class="PartnerLink">
   partnerRole="PR1"                   <property name="partnerRole" value="PR1"/>
   partnerLinkType="PLT1"/>            <property name="partnerLinkType" value="PLT1"/>
<variable name="V1"                    </bean>
   messageType="MT1"/>                 <bean id="V1" class="Variable">
<variable name="V2"                    <property name="messageType" value="MT1"/>
   messageType="MT2"/>                 </bean>
<invoke name="I1"                      <bean id="V2" class="Variable">
   partnerLink="PL1"                   <property name="messageType" value="MT2"/>
   portType="PT1"                      </bean>
   operation="OP1"                     <bean id="I1" class="Invoke">
   inputVariable="V1"                  <property name="partnerLink" ref="PL1"/>
   outputVariable="V2"/>               <property name="portType" value="PT1"/>
                                       <property name="operation" value="OP1"/>
   BPEL                                <property name="inputVariable" ref="V1"/>
   Spring                              <property name="outputVariable" ref="V2"/>
                                       </bean>
```

Figure 6.2: Spring configuration for a BPEL specification.

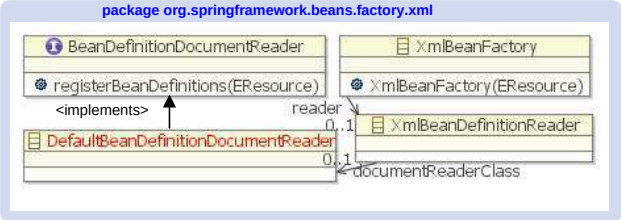


Figure 6.3: The class modified for underlying extension.

6.4.1 Component - A

This is the core component of the framework that renders a bean-factory out of a BPEL specification. It is based on the Spring framework and thereby offers all its benefits (e.g. loose coupling and dependency injection).

The similarity between a BPEL specification and a Spring configuration file forms the basis of this component. Although structurally dissimilar, they both contain the requisite information to instantiate and initialise Java beans. Consequently the Spring framework has been extended to parse a BPEL specification and 1) recognise the BPEL activities 2) fetch all information associated with an activity 3) instantiate and initialise appropriate Java beans. This is illustrated using Figure 6.2 wherein an equivalent Spring configuration has been formulated for a BPEL specification. The extended Spring framework operates identically with both the XML files and render identical Java beans. Figure 6.3 highlights the class that was modified to implement this feature. The default logic for parsing an XML configuration file is implemented in *registerBeanDefinitions* method of *DefaultBeanDefinitionDocumentReader* class. We extended the default logic to allow parsing a BPEL specification. The two steps

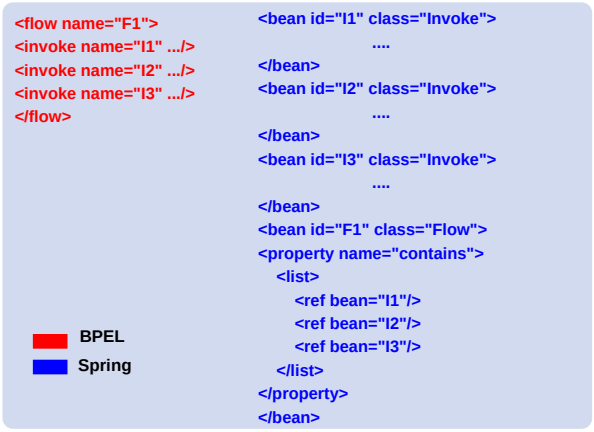


Figure 6.4: Bean for structured activities store reference to child activities.

associated with this component are now discussed.

Step 1. Instantiation

The first step involves parsing a BPEL specification and instantiating the appropriate Java beans. When parsing a BPEL specification, the extended framework initially fetches all the XML elements in it. Thereafter it looks for classes corresponding to each of these elements. Considering that each BPEL activity is mapped to a Java bean, the framework should find an appropriate class for each element. The class names are essentially same as the mapped activity names (with the exception of first character that is in upper-case). Usually a Java bean has one property per attribute of the corresponding activity. The framework uses the 'setter' methods to initialise these properties with appropriate values. However, the Java bean for BPEL structured activities have an additional property to store the references to underlying child activities. For instance, as shown in Figure 6.4, the bean for *flow* activity would store the references to all its child activities.

Instead of manually creating the Java beans, we generate them automatically using the Eclipse Modeling Framework (EMF) [emf, 2010]. The framework offers a graphical interface to create an object model that is used to generate the classes. Figure 6.8 illustrates the object model that is used to create the Java beans for the proposed verification framework. All attempts have been made to introduce hierarchy into the object model. Considering that a large number of BPEL activities have identical attributes, this allows a child class to reuse the properties of its parent class. The object model is further explained in Section 6.4.3.

Step 2. Dependency Injection

After instantiating the Java beans in previous step, their properties need to be initialised. In this step, the framework uses the appropriate 'setter' methods to assign these properties. Considering that this resolves any dependency between collaborating objects, the process is also known as *Dependency Injection*.

As illustrated in Figures 6.2 and 6.4, a property might be assigned either a value or a reference to another bean. While a value is immediately available, a reference might not be available until the corresponding bean is instantiated. For instance the property *contains* of *Flow* class in Figure 6.4 can only be initialised after all the three *Invoke* beans have been instantiated.

At the close of second step, the framework renders a bean-factory. This factory allows retrieving the containing objects by their name.

6.4.2 Component - B

This component yields a formal model based on the bean-factory and an XML schema. As discussed previously, it has two subcomponents: 1) a binding compiler and 2) a binding runtime. The binding compiler requires an XML schema or a DTD that defines the structure of the envisioned XML document. It creates a set of classes that conforms to this structure. These classes needs to be instantiated and initialised before the binding runtime can use them to generate the required XML document. The objects from the bean-factory are used to initialise these classes. The two steps associated with this component are now discussed.

Step 3. Bind

In this step, the binding compiler generates a set of annotated classes based on a schema or DTD. This schema determine the structure of the formal-model produced and is different for each solution proposed. Considering that most researchers prefer to propose their solution as a set of XML models (rather than the corresponding schemas) [Kang et al., 2007; Yi and Kochut, 2004; Yang et al., 2005; Sloan and Khoshgoftaar, 2009; Ouyang et al., 2007], the tool *Trang* [tra, 2011] can be used to obtain their XSD schemas. Usually a schema is required per BPEL activity to transform the corresponding Java bean into a model. Figure 6.5 illustrates an excerpt from a CPN model (which is XML based) and the corresponding schema obtained using Trang. The auto-generated schema can be customised to further streamline the formal model rendered by the component.

Figure 6.5: The schema for a fragment of CPN model.

Step 4. Marshal

In this step, the binding runtime actually produces the formal model from the classes generated in Step 3. However, as a pre-requisite, these classes must be instantiated and initialised using the objects in the bean-factory. Considering that these classes have getter and setter methods for each property to be initialised, they are instantiated using a Spring configuration file wherein their dependencies are injected to produce another bean-factory. Please note that this bean-factory is different from the factory generated by Component - A and its beans are restricted to the classes generated in Step 3. After initialisation, the objects in this bean factory are used by the binding runtime to generate the formal model. The instantiation and initialisation of classes are further explained using Figures 6.6 and 6.7.

Figure 6.6 illustrate a CPN based solution wherein a place is initialised with all partner-links in a BPEL document. Consequently this requires copying the properties of *PartnerLink* beans into an appropriate section of the CPN excerpt (shown in brown). However, since this CPN excerpt is to be automatically generated by JAXB bind runtime, the classes used by it needs to be initialised with this value.

The process of initialising the classes is explained using Figure 6.7. When the schema corresponding to the CPN excerpt is compiled (using xjc), each element of the excerpt renders a new class. Furthermore these classes procure properties per attribute and per child node of the corresponding element. Considering that the node to be initialised is the child of *text*

```
<place id="ID1003057409">
 <posattr x="-298.000000"
     y="19.000000"/>
 <text/>
 <type id="ID1003057410">
  <text tool="CPN Tools"
     version="2.2.0">
     PARTNERLINK</text>
 </type>
 <initmark id="ID1003057411">
  <text tool="CPN Tools"
     version="2.2.0">
       1`("PL1","PLT1","MR1","PR1")++
       1`("PL2","PLT2","null","PR2")
  </text>
 </initmark>
</place>
```

```
Class PartnerLink{
    String name;
    String partnerLinkType;
    String myRole;
    String partnerRole;
  //getters and setters
}
```

CPN excerpt
Java Bean
initialise from bean

Figure 6.6: A CPN excerpt wherein the initial marking of a place is assigned from Java beans.

```
public class Place {
  protected Initmark initmark;
        ....
  //getters and setters }

public class Initmark {
  protected Text text;
  protected String id;
  //getters and setters }

public class Text {
  protected String content;
  protected String tool;
  protected String version;
  //getters and setters }

class Initialise{
  private Text text;
  public void setBf(BeanFactory bf){
    PartnerLink pl=
        bf.getBean("partnerLink");
    text.setContent(pl.getName+...);
  }
  //getters and setters }
```

```
<bean id="place" class="Place">
  <property name="initmark" ref="inim"/>
                ....
</bean>
<bean id="inim" class="Initmark">
  <property name="text" ref="text"/>
                ....
</bean>
<bean id="text" class="Text">
  <property name="tool" value=".."/>
  <property name="version" value=".."/>
</bean>
<bean id="file" class="org.springframework.
                core.io.FileSystemResource">
  <constructor-arg value="bpel.xml"/>
</bean>
<bean id="ft" class="org.springframework.
        beans.factory.xml.XmlBeanFactory">
 <constructor-arg ref="file"/>
</bean>
<bean id="beanSecond" class="Initialise">
 <property name="bf" ref="ft"/>
  <property name="text" ref="text"/>
</bean>
```

Figure 6.7: The instantiation and initialisation of classes for CPN excerpt in Figure 6.6.

element, it should be assigned using the *content* property of *Text* class. However, the value to be assigned is stored in the bean-factory generated by Component - A. Consequently this bean-factory is fetched (using XmlBeanFactory) and injected into *Initialise* class to assign the value. As shown in Figure 6.7, each bean whose properties need to be assigned is also injected into this class. Since the bean-factory from Component - A is injected as a property *bf*, the method *setBf* is invoked for initialising the properties of other beans. It is worth mentioning that 'BPEL.xml' in Figure 6.7 is the BPEL specification document.

We now introduce the object model for BPEL activities. Apart from being the basis of

BPEL activity classes, the proposed XML templates are also based on it. Having a standard object model helps in customizing the templates on the fly. This is because the places to look for information in each case is known.

6.4.3 The Object Model for BPEL Activities

This section illustrates the semantic and syntactic relationship among BPEL activities using Figure 6.8. For instance the activities *invoke*, *receive* and *reply* are subclasses of a parent class *InterfaceActivities* owing to their semantic similarity. Similarly due to the syntactic similarity, the class *Activities* has properties *suppressJoinFailure* and *joinCondition* that are common for its child classes. Using such relationships, a generic template can be proposed for a parent class, that could then be customized for each of its child classes. Each activity in Figure 6.8 is represented as a class that is related to other classes using either 'is-a' (inheritance) or 'has-a' (composition) relationships.

The following discussion outlines the relationship among classes in Figure 6.8. These relationships are reflected in the XML templates introduced in the next section.

At the top of the hierarchy is the class for *Process* that forms the root of any BPEL document. It contains a reference to *GlobalScope* that signifies the top-level scope in a BPEL process. The class *GlobalScope* in-turn has references to *LocalScope* for each secondary-level scope. Considering that a scope in a BPEL specification can have nested-scopes with arbitrary depth, the class *LocalScope* has a self-reference in order to identify the parent-scope. For instance the ternary-scopes have the secondary-scopes as their parent.

Based on this relationship among global and local scopes, we have decided to incorporate hierarchical CPNs to model a BPEL specification, wherein each additional level of hierarchy corresponds to a supplementary depth of nested scope. This reiterates the importance of utilizing these relationships when formalizing a BPEL specification.

A scope can contain variables, compensation-handlers, fault-handlers, event-handlers and a primary activity. Table 6.2 lists the class for each of these activities. The class *Variable* has properties that store the name and contents of a variable. In addition, it also has an entry to store the variable type. The enumeration *VariableTypes* contain all 'types' for BPEL variables. The class for primary activity and those for each of the handlers contain one or more references to 'Activities'. Consequently they are discussed after introducing the classes for BPEL activities. These relationships are reflected in the template for local and global scopes that are proposed in following section.

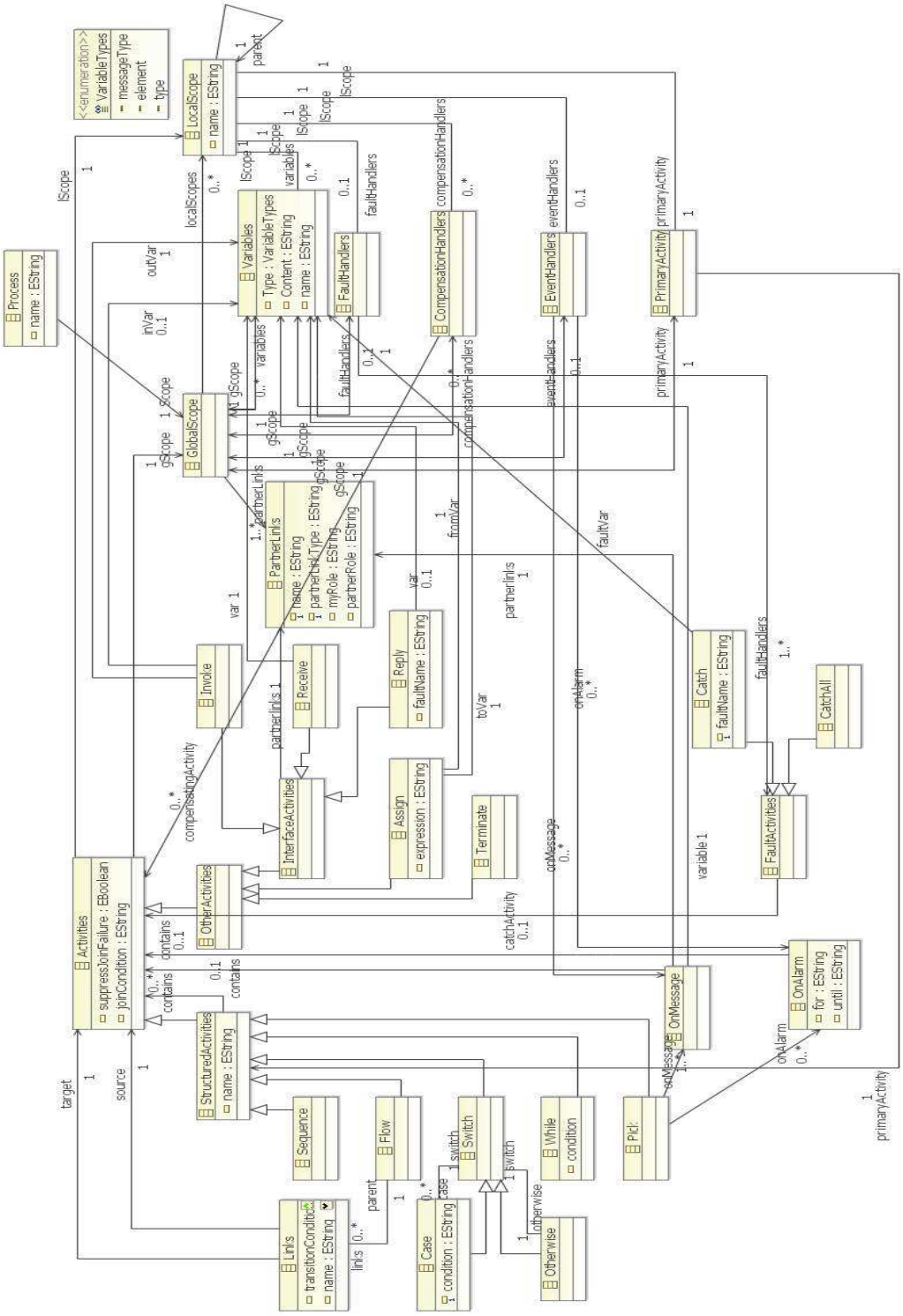


Figure 6.8: The Object Model for BPEL Activities

Table 6.2: The classes for compensation-handlers, fault-handlers, event-handlers and primary activity

BPEL Activity Name	**Class Name**
faultHandlers	FaultHandlers
eventHandlers	EventHandlers
compensationHandlers	CompensationHandlers
Primary activity of a scope	PrimaryActivity

Table 6.3: The classes for BPEL structured activities

BPEL Activity Name	**Class Name**
sequence	Sequence
flow	Flow
switch/case/otherwise	Switch/Case/Otherwise
while	While
pick	Pick

The class *Activities* forms the superclass for all BPEL activities. It has references to *GlobalScope* and *LocalScope* that store the top-level and the enclosing scopes. All activities in the top-level scope have their *lScope* assigned to *null*. Considering that a BPEL specification has a single top-level scope, all activities contain the same reference in *gScope*. In addition, it has two properties that are essential to express synchronisation dependencies using the BPEL link activity. These properties are included in the superclass as any BPEL activity can be the source or target for a link. The references *source* and *target* in *Link* stores the references to these activities.

Based on these relationships, the next section proposes two XML templates corresponding to an activity. While the first template is applicable for activities that are synchronized by links, the other template is used for activities that are neither the source nor the target of a link.

The class *Activities* has two subclasses, *StructuredActivities* and *OtherActivities*. As the names indicate, former is the superclass for all structured activities in BPEL, while the remaining activities extend latter. The list of sub-activities of a structured activity is represented by a reference *contains*. This reference to *Activities* is included in superclass because all structured activities in BPEL can have sub-activities. Table 6.3 lists the classes for structured activities.

The class *Flow* has a reference to *Links* that stores all the links defined within a flow-activity. The class *Links* also has a reference to *Flow* that stores the enclosing flow activity

as its parent.

The class *Switch* contains references to one or more instances of *Case* that also happens to be its sub-class. The *Case* class has a property *condition* that is evaluated by *Switch* for each of its instances. The first instance encountered that satisfies the condition has its sub-tasks executed. Additionally it also contains a reference to its sub-class *Otherwise* whose sub-tasks are evaluated when none of the conditions are satisfied.

The *While* class corresponds to BPEL while activity. It has a property *condition* that causes the sub-tasks to be executed repeatedly until it evaluates to false.

The class *Pick* corresponds to BPEL pick activity. Considering that a pick activity can have any number of onMessage and onAlarm activities (at least 1 onMessage activity), the class *Pick* contains reference to *OnMessage* and *OnAlarm* instances. The class *OnMessage* has a reference to *PartnerLinks* owing the partnerLink attribute of onMessage activity. It also has a reference to *Variables* corresponding to variable attribute. The *OnAlarm* class stores the duration and deadline in its *for* and *until* attributes.

The classes *Invoke*, *Receive* and *Reply* extend a common class *InterfaceActivities*. As the name suggests, *InterfaceActivities* is the superclass for all BPEL activities that are involved in interaction with other web-services. Consequently it contains a reference to *PartnerLinks*. The class *Invoke* has references *inVar* and *outVar* corresponding to inputVariable and outputVariable attributes of invoke activity. Similarly *Receive* and *Reply* contain a reference *var* corresponding to the attribute variable. In addition, *Reply* has a property *faultName* to store the name of the fault in case of synchronous operation.

The class *Assign* corresponds to BPEL activity assign. It is used to copy values between variables and expression. Consecutively it has a property *expression* and two references *toVar* and *fromVar*.

The handlers listed in Table 6.2 are now discussed. The class *FaultHandlers* contain references to *FaultActivities* which in turn contains a reference to *Activities*. While the reference in *FaultHandlers* stores instances of underlying *Catch* and *CatchAll* classes, the reference in *FaultActivities* hold the sub-activities of a *Catch* or *CatchAll* activity. The *Catch* class has a property *faultName* to store the fault-name and a reference *faultVar* to store the fault-variable. Based on the fault, a *Catch* or *CatchAll* instance is selected and its sub-activities are executed.

The class *CompensationHandler* contains reference to a set of *Activities* instances that are executed when the BPEL 'compensate' activity is invoked for a particular scope. The reference *parent* in a scope is used to determine its enclosing scope and to ensure that the

compensate activity is present in the compensation-handler or fault-handler of this enclosing scope.

The class *EventHandlers* contain reference to the underlying *OnMessage* and *OnAlarm* instances that in turn contain reference to the list of sub-tasks to be executed in the event of receiving a message or a time-out. When compared to the *Pick* class, the *onMessage* reference in *EventHandlers* has a lower-bound of '0'. This follows from the fact that a BPEL pick activity must have at least one underlying onMessage activity while event-handlers might have none.

Each BPEL scope has a primary activity that is stored in *primaryActivity* reference of a *PrimaryActivity* instance. Primary activities are often structured activities containing one or more underlying sub-activities. Even when primary activity is not structured, it can be wrapped with a BPEL sequence activity. Consequently the reference *primaryActivity* contain instances of *StructuredActivities*.

The proposed XML templates reflect the aforementioned identified relationships. Considering that

- *GlobalScope* (and *LocalScope*) 'has-a' *FaultHandlers*
- *GlobalScope* (and *LocalScope*) 'has-a' *EventHandlers*
- *GlobalScope* (and *LocalScope*) 'has-a' *CompensationHandlers*
- *GlobalScope* (and *LocalScope*) 'has-a' *Variables*
- *GlobalScope* (and *LocalScope*) 'has-a' *PrimaryActivity*

these activities are included in the templates for Global and Local scopes in the next section. This allows each scope to have its exclusive set of handler, variable and primary activities.

6.4.4 The Proposed XML Templates

The XML templates for formalizing BPEL activities are proposed in this section. These templates (i.e. their schemas) are used by JAXB 2 compiler to create a formal model using the Java beans. As mentioned previously, the templates are based on object model defined in previous section.

In order to model both the data and control flows, the BPEL activities are formalised using Coloured Petri nets (CPN). In a CPN model, 1) the firing of transitions are used to depict the control flow between places along the arcs, and 2) the transfer of tokens among

places is used to depict the data flow. The flow for BPEL activities have been accordingly mapped in the proposed templates.

Template for Global-Scope

In this section we present the template that forms the basis of any formal-model. Additional information from the object-model is used to customize this template to obtain the formal model.

A CPN model could be constituted of either a single page or a number of pages. While the former produces a flat representation, latter produce a hierarchical representation. Considering that the scopes in a BPEL specification could also be flat or hierarchical, we use a separate page to represent each scope. The top-level (global) scope is imperative in all BPEL specifications and the absence of any other scope result in flat orientation. The presence of any additional scopes result in a hierarchy.

In this section, the template for a page corresponding to the top-level scope is discussed. Considering that any formal-model would have this page, it is used as the base in constructing the model. The template for pages corresponding to additional scopes is discussed in the next section.

Figure 6.9 illustrates the proposed XML template for top-level scope. It also depicts the model obtained when the template is opened with CPN Tools. Only the essential elements of the template are shown because of the space limitation.

The DTD for CPN tools require the template to have <*workspaceElements*> and <*cpnet*> as its outermost elements. The element <*globbox*> contains the declarations for colour-sets, functions and variables. Being the base template, it contains declaration for all other templates that might be used later. This relieve us from recording the declarations in each template and including them when the template is used.

The DTD require the templates to have a <*page*> element for each page in the CPN model. As discussed earlier, the base template has a single page corresponding to the top-level scope. Its child element <*pageattr*> contains an attribute that assigns the page a name. Each <*place*> element thereafter render a place in the corresponding page of the model.

The elements that can appear more than once have an attribute 'id' in order to recognize them. For instance each <*page*> has a unique id that helps in identifying and differentiating it. The DTD require that no two elements in the template have same id. Consequently a counter is maintained to assign unique ids to each element as they are added. Each id should

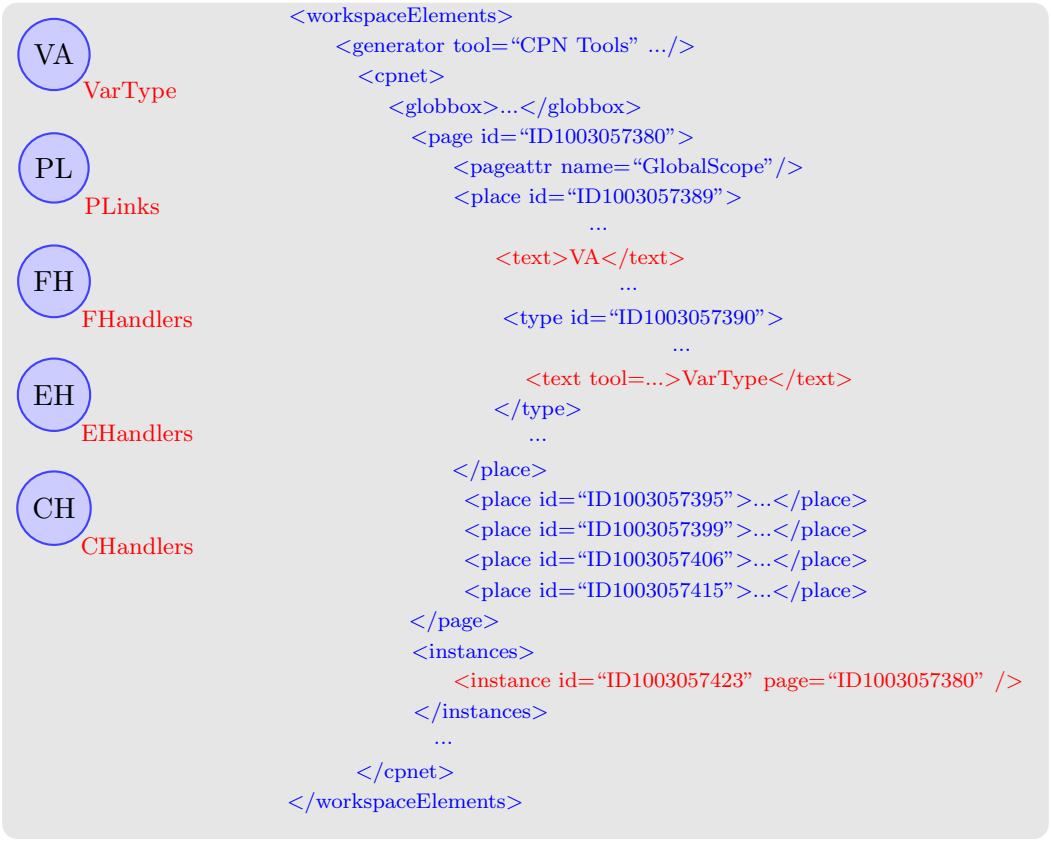


Figure 6.9: The template for top-level scope

be an integer preceded by 'ID'.

The five <*place*> elements in the XML template correspond to the five places shown in Figure 6.9 and they appear in same order. The <*place*> element for topmost place is illustrated in detail to explain their relationship. The text 'VA' appearing inside the place comes from the sub-element element <*text*> that is highlighted in red. Similarly the colour 'VarType' is fetched from the sub-element <*text*> of sub-element <*type*> which is also highlighted in red. The declaration for colour-set *VarType* could be found within <*globbox*> element. Other place elements are defined similarly but are collapsed due to space constraints.

Finally the element <*instance*> instantiates a page. Unless a page element has an <*instance*> entry within <*instances*>, the page do not appear in the model. Consequently new entries would be added here for each new page included into the model.

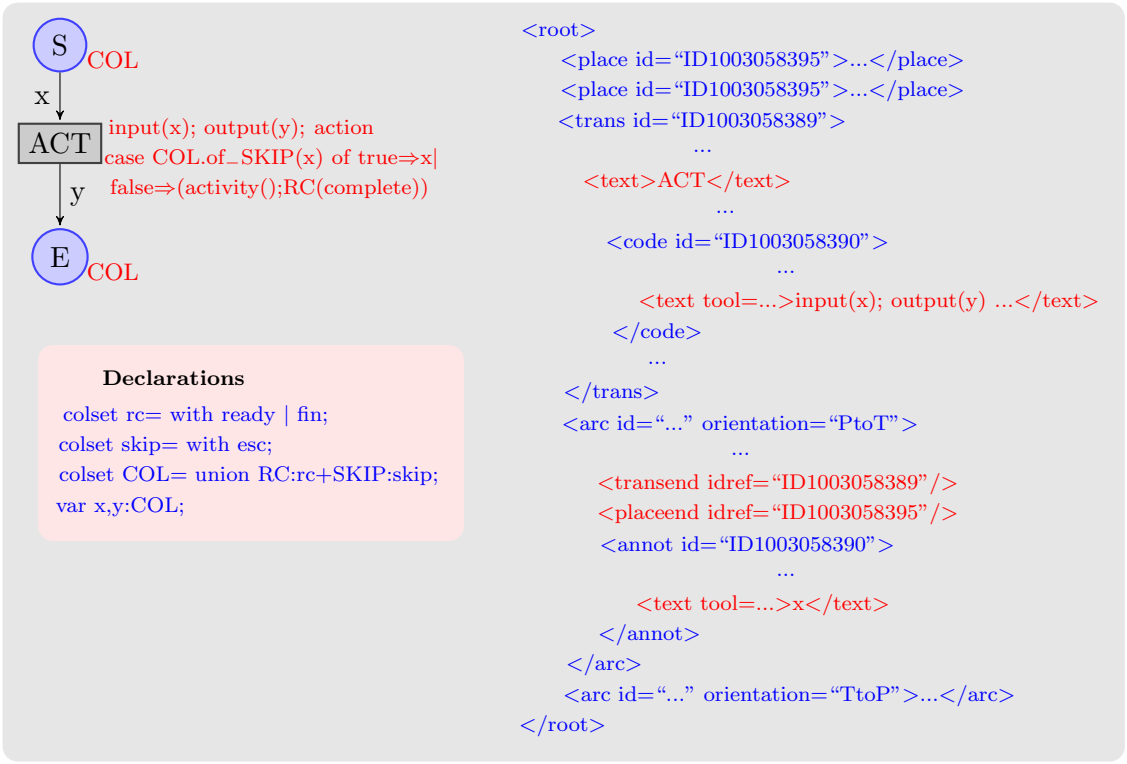


Figure 6.10: The template for a basic activity

Template for an Activity

In this section, the template for a basic activity is introduced. This template can be used for any basic activity with some activity-specific customizations that are discussed later. In this context, an activity is considered basic if it is neither the source nor the target of a link activity.

Figure 6.10 illustrates the XML template for a basic-activity. Before an activity starts executing, it should be "ready" for execution. Similarly once it executes, it should have "finished" execution. These two states of a BPEL activity are stored in CPN tokens as *ready* and *fin* as shown in Figure 6.10. Some activities (e.g. invoke) might not finish immediately and could have a waiting time. As illustrated later, such activities are modeled using *timed CPN* by adding a delay into transition *ACT*.

Assigning a token with the aforementioned values model the normal execution of an activity. However, in case of dead-path elimination, an activity might have to be skipped completely. In order to model such scenarios, the token is assigned a value 'esc'. The

transition ACT examines the value in token and behaves accordingly. Such a selective behaviour requires programming the *code segment* of the transition as shown in Figure 6.10.

The values that a token should have for normal execution are declared as an enumeration *rc*. Similarly the value for skipping are defined in *skip*. In order to ensure that the token can be assigned values from either of these enumerations, the containing places are assigned a colour-set that is the union of these colour-sets. This union colour-set is assigned to places S and E in Figure 6.10. In this context, former acts as a start place while the latter acts as a finish place. A start place contains a *ready* token when an activity is ready to execute. Similarly a finish place contains a *fin* token after the activity finishes execution. In order to skip an activity, the start place is populated with a token *esc*.

Considering that an activity behaves differently based on the token in S, a function cs.of_col() is used to dynamically evaluate them. This function is natively defined for CPN tools. For a union 'u' composed of colour-sets 'c1' and 'c2', u.of_c2(v) returns true only if the token 'v' contains a value of type c2. The transition ACT uses this function to decide on the outcome of its execution. It accepts the token in place S and checks its value. If the token contains a value from *skip* colour-set, the activity is not executed. Otherwise the activity is executed by calling the function *activity()* and a token with value *fin* is forwarded to place E.

The code-segment of a transition is contained within a <*code*> element as shown in Figure 6.10. The template also contain two <*arc*> elements corresponding to the two arcs. The attribute *orientation* denotes if an arc is directed from a place-to-transition (PtoT) or otherwise. Its sub-elements <*transend*> and <*placeend*> contain the ids of transition and place that it connects. Furthermore, the arc inscriptions are contained within the sub-element <*annot*>.

Template for an Activity Synchronized by Links

In this section, we introduce the template for an activity that is synchronized using links. Considering that links are always defined within a *flow activity*, the actual synchronization is discuss later when we discuss the template for latter.

Figure 6.11 illustrate an activity whose synchronization dependencies are expressed using links. The template from previous section is used to model the activity. Such a practice of reusing templates lead to reduction in size of the new template. This reduction is attributed to the expulsion of any element common across several templates.

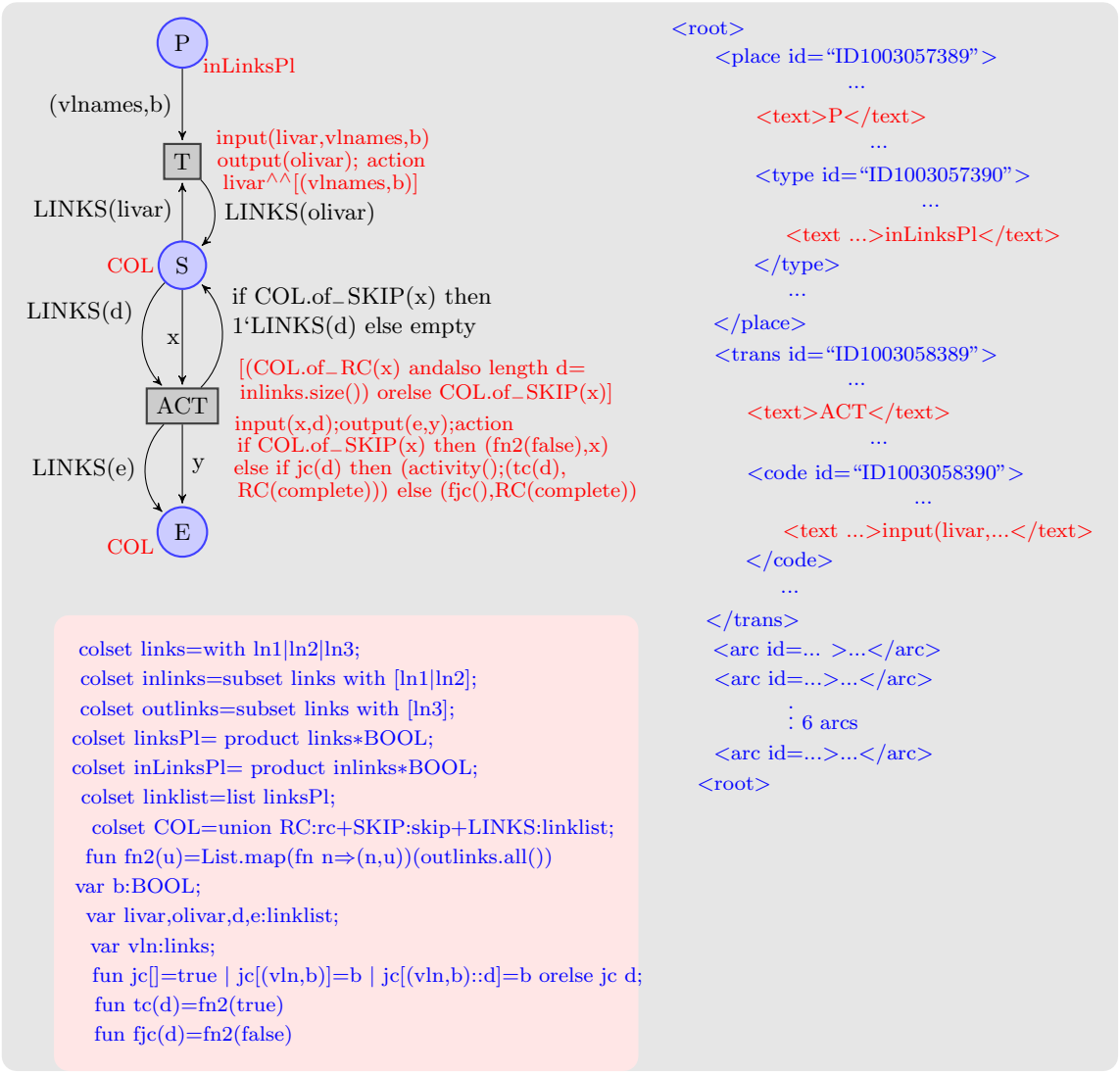


Figure 6.11: The template for an activity synchronised by links.

Each link has a *source* and *target* activity. The links that have ‘A’ as its target activity form a colour-set $inlinks_A$. Similarly the links that have it as source activity form another colour-set $outlinks_A$. These colour-sets are subsets of $links_A$ that contain all the links associated with ‘A’. These sets for ACT is shown in Figure 6.11 (subscripts omitted).

Each incoming link has an associated boolean status that is stored in place P along with the link-name. The place S initially contains an empty list of such link-name and status pairs. As the status of each link is available in place P, they are added to this list by transition T. When the status for all incoming links have been added to it, its length must be equal to the

number of elements in colour-set *inlinks*. This causes transition ACT to execute and remove the list along with the ready token. In order to prevent the transition from executing until the status of all incoming links are available, a guard condition is attached to it.

As with a basic activity, the token is assigned a value *esc* in order to skip the activity. The transition is programmed to customize its behaviour based on the contents of token in S.

Considering that an activity might be the source of one or more links, the transition ACT needs to evaluate their *transition conditions* (if any) and determine their status. The code segment of the transition is programmed for this purpose. Initially the *join condition* is evaluated based on the status of incoming links. The function *jc()* in Figure 6.11 performs a logical OR operation on the status of all incoming links. This is also the default behaviour if no condition is specified. If this function returns true, the activity executes and the transition conditions of each outgoing link is evaluated. Otherwise the activity is skipped and the status of each outgoing link is assigned false. In either case a token with value *fin* is sent to place E. The model provides dummy implementation of these functions (e.g. tc() for transition conditions) and they can be customized based on requirement.

Template for Local-Scopes

In this section, the template for an underlying scope is proposed. As discussed previously, the proposed model represent each scope in the BPEL specification using a separate page. Consequently the template has two sub-templates, 1) the template for substitution transition in the superpage (i.e. the page for parent scope) and 2) the template for page that is added for each new scope.

Figure 6.12 illustrate the template for the substitution transition that is added into the page for a parent scope for each of its child scopes. Considering that a scope might need to use the variables, fault-handlers, event-handlers and partnerlinks (if parent is top-level scope) of the parent scope, a bidirectional arc connect each of the corresponding places to the substitution transition. Like any other activity, the model for scope has place S and E.

In addition to the elements that exists in a regular transition, a substitution transition has an element *<subst>* that contains, among other things, the ID of page it replaces and the port-socket mappings. Consequently the XML template for the substitution transition consists of this single distinguishing element. It is appended to the template for a regular transition to obtain the complete template for the substitution transition. This is in

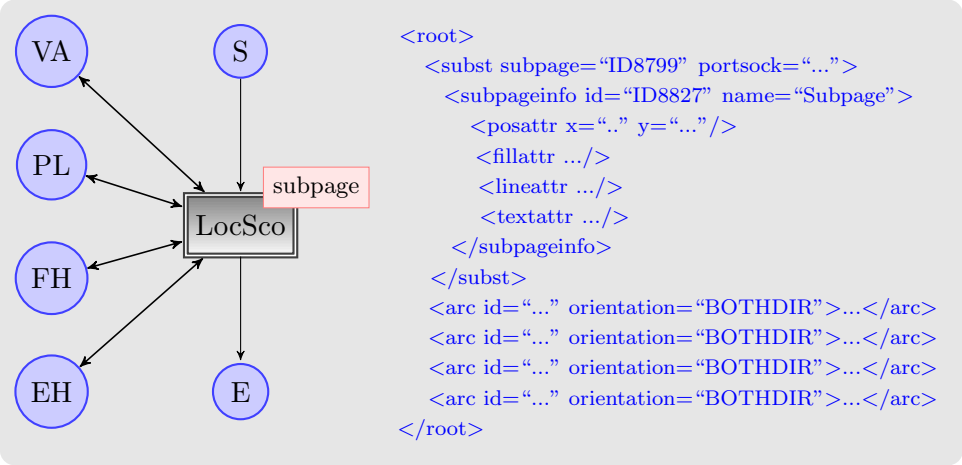


Figure 6.12: The template for items added to parent page.

accordance with the previously discussed principle of template-reusability.

Figure 6.13 illustrate the template for a page corresponding to an underlying scope. It is similar to the template for global scope and has places to store the variables, fault-handlers and event-handlers that are defined in the corresponding scope. In addition it has port-places that allow it to access the variables and handlers from overlaying scopes. This allow modeling a deeply nested scopes without any bounds.

When the substitution transition corresponding to a scope execute, the token from place S in superpage is moved to its port-place in subpage (also named S in this case). Thereafter the activities in this scope execute and consume the tokens. Finally the leftover token in port-place E is copied back to its socket place.

Template for Flow Activity

In this section, we propose the template for flow activity. As indicated previously, this section also considers specification of synchronization dependencies using links. Initially the arcs, places and transitions in red are ignored. The places S and E along with their colour-set ensure the compatibility of flow with the proposed templates. Furthermore it has a normal execution and a skipped execution like other activities. In case of normal execution, it simultaneously adds a *ready* tokens into the starting places of each sub-activity, causing them to execute in parallel. It then waits until each of its sub-activities finish execution by collecting the *fin* token from their *finish* places. Thereafter it quits by populating its finish place E with a *fin* token.

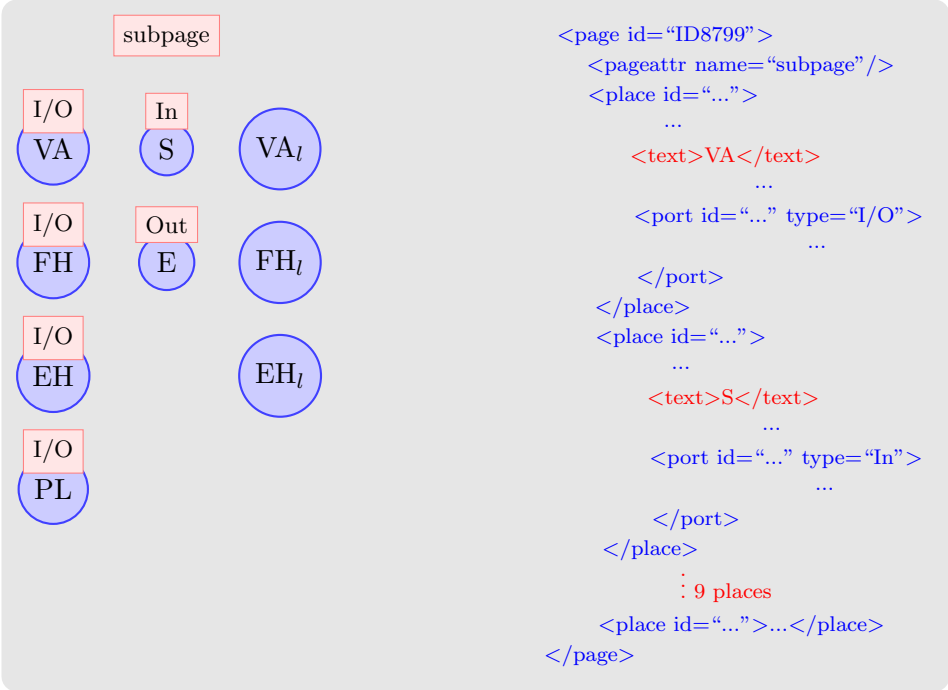


Figure 6.13: The template for subpage corresponding to an underlying scope.

The template for flow consists of a transition with an incoming and an outgoing arcs. This template is used in conjunction with the previously proposed templates to instantiate a flow activity. For instance a minimal flow activity would contain a single sub-activity (say A1). In order to model it, the places S and E along with the model for activity are created using place and activity templates. Thereafter the template for flow is used to add a transition between S and S_1, E_1 and E & S and E. Adding additional sub-activities would require two arcs that can be drawn using the template for arcs.

The specification of synchronization dependency is illustrated using places, transitions and arcs in red. The template proposed for *link* is used for this purpose. In addition, the flow template is used to connect E_3 and P using T_0. We consider a single link that has *A3* as its source and *A2* as target. Consequently the former must finish execution before the latter starts. As pointed out for *link*, the place E_3 would be populated with a list once *A3* terminates. This list would contain (link-name,status) pair for each link that has *A3* as its source. In this case it would contain the status for sole link considered. The transition T_0 breaks this list and populate place P with the individual elements in it. Thereafter the status is passed to *A2* which executes selectively based on the status of the link.

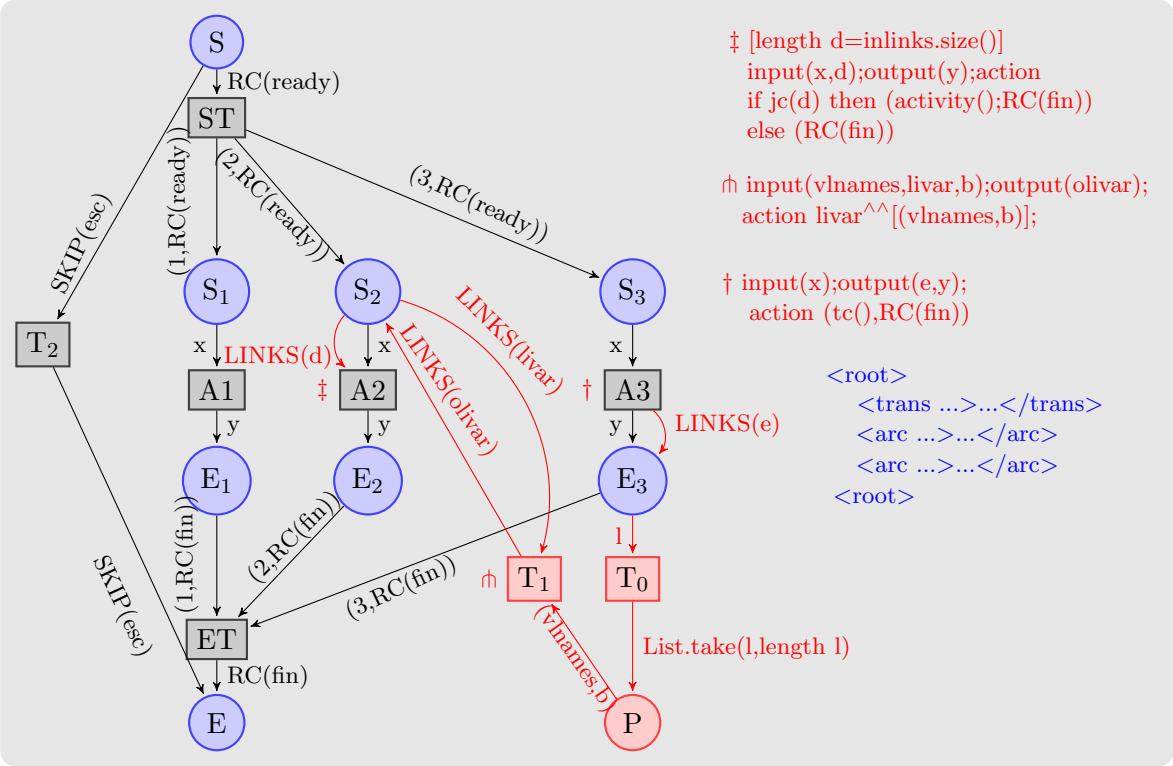


Figure 6.14: The template for flow activity.

It should be noted that the template for *link* was proposed for an activity that had both incoming and outgoing links. However, the activities *A2* and *A3* in Figure 6.14 have either incoming or outgoing links. Consequently the template has been dissected into two parts for modeling these activities.

Template for Sequence Activity

In this section, the template for sequence activity is proposed. It is usually the primary activity of top-level scope in a BPEL specification.

Figure 6.15 illustrates the template for sequence activity. At first look, it seems to be similar to the template for flow. However, the templates differ in the expression attached to input and output arcs.

A sequence activity ensures that its sub-activities execute in the specified sequence. Consequently when modeling it, it is sufficient to populate the place S of an activity with *ready* token once its previous activity has *fin* token in E. This would allow an activity to begin

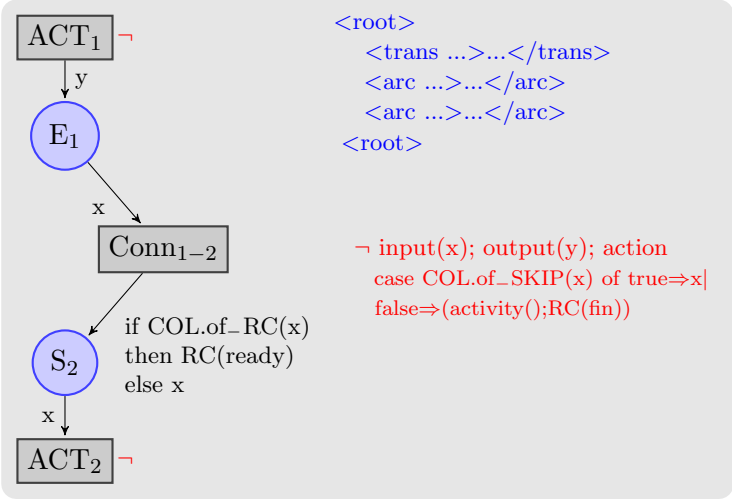


Figure 6.15: The template for sequence activity.

execution once its previous activity has finished execution.

Additionally, in case of Dead-Path elimination, an activity might need to skip if its previous activity was skipped. In order to model such scenarios, the sequence template behaves selectively and populates the place S of an activity with *esc* token when it finds a similar token in place E of previous activity.

Template for Switch Activity

In this section, the template for switch activity is proposed. It allows multi-way conditional branching and is used to introduce decision points in order to control the flow of a BPEL process.

Figure 6.16 illustrates the template for switch activity. As with sequence, the template for switch looks similar to flow activity. Such a similarity is deliberate as it enhances the re-usability of a template. However, the arc expressions and code segments differ from both flow and sequence activities.

In order to fit into the template for a basic-activity, it contains the required start and finish places (i.e. S and E). In addition, it has a place D to contain the token based on which branching decision is finalised. The transition and arcs between S & D and D & E are created using the switch template.

The transition *Decision* inspects the token in S to check if it is a normal of skipped execution. For normal execution, the function *getCase()* is used to decide on the case to be

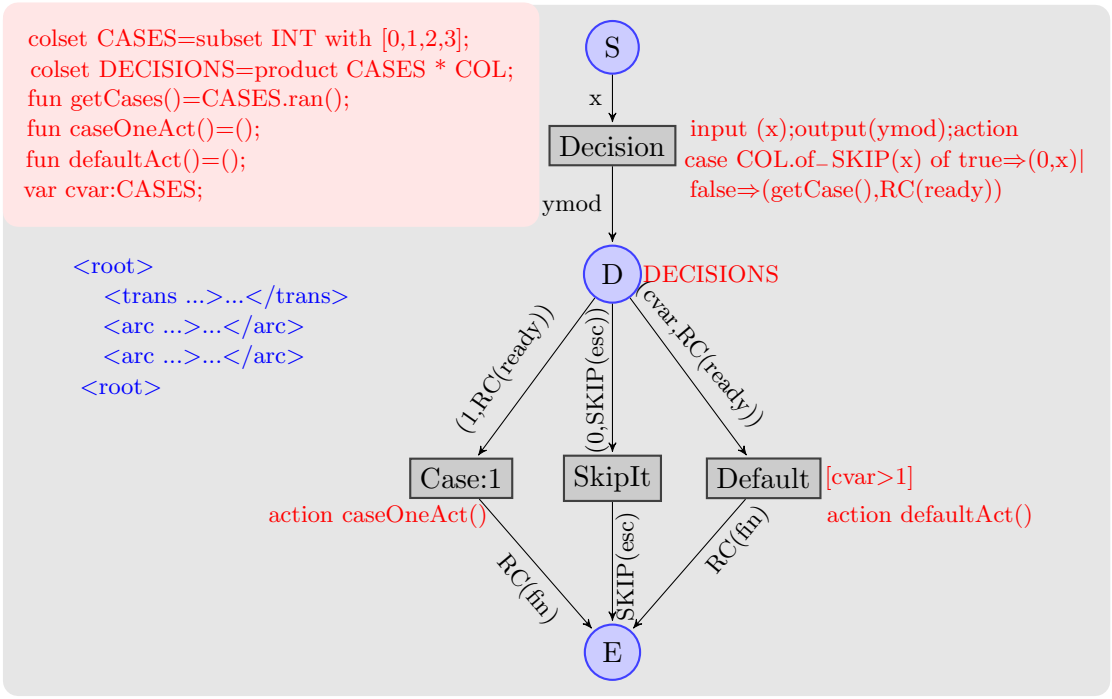


Figure 6.16: The template for switch activity.

executed. By default the function is implemented to select a case randomly.

Each case is assigned a unique case-number and *getCase()* returns the case-number for selected case. However, *default* case is assigned special case-numbers. As shown in Figure 6.16, any case number greater than the number of cases would cause the default case to execute. This requires assigning the case-numbers sequentially. The code-segment for each case contains a function caseXxxAct() that executes the underlying activity.

The case of skipping an activity is assigned a dummy case-number of 0.

Template for Invoke, Receive and Reply Activities

In this section the template for interface activities are discussed. As pointed out earlier, these activities often have a waiting time as they are used to communicate with other web-services.

Each partner web-service for a BPEL process is specified as a *partner-link* in the specification. These are used by the interface activities to 1) call an operation (< *invoke* >), 2) offer an operation and wait for it to be called (< *receive* >) or 3) return the result of an operation (< *reply* >). The parameters for these operations are modeled as messages that are stored in variables. Consequently these activities need access to the required variables.

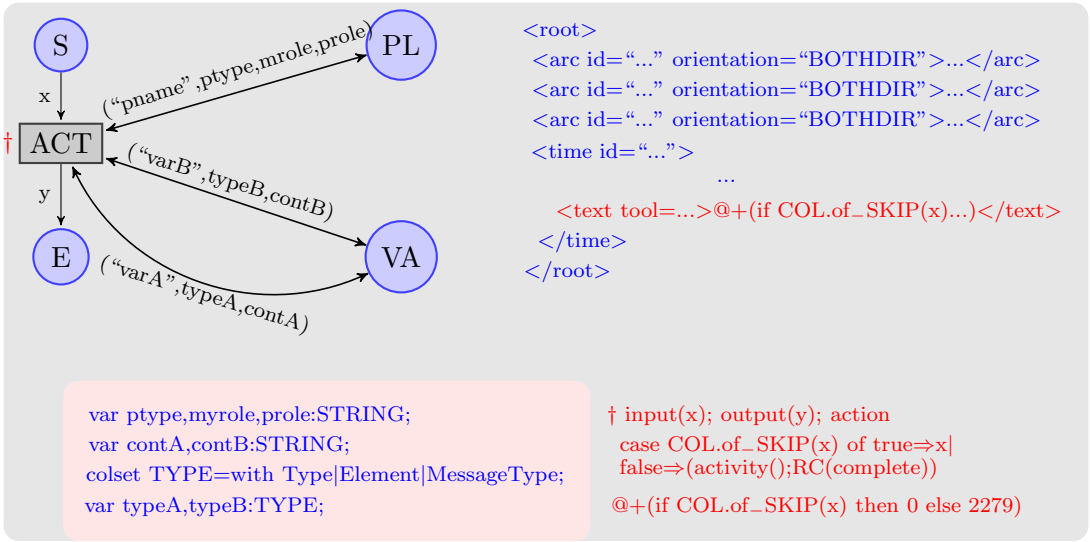


Figure 6.17: The template for inteface activities.

Table 6.4: The use of three additional arcs in interface template

BPEL Act	**Arcs**		
Interface Act	pname	varA	varB
invoke	access partner-link token	output variable	input variable (synchronous operation)
receive	access partner-link token	-	input variable
reply	access partner-link token	output variable	-

Figure 6.17 illustrates the template for interface activities. The additional arcs are used to access the required partner-links and variables. Although they always need a single partner-link, they might need to use more than one variable. For instance $< invoke >$ uses both input and output variable in case of synchronous operation. Table 6.4 illustrates the number of variables used by each of the activities.

The time delay in executing an interface activity is modeled using timed CPN. This requires attaching a time delay to the transition ACT by adding a $< time >$ element to the transition template. This element is a part of the template for interface activities.

In timed CPN, each token has an associated time-stamp. When a transition fires, the time-stamp is incremented by the delay specified in the transition. The model in Figure 6.17 specifies a random delay for transition ACT. Specifying the delay allows in accounting for the time incurred in interacting with a partner web-service.

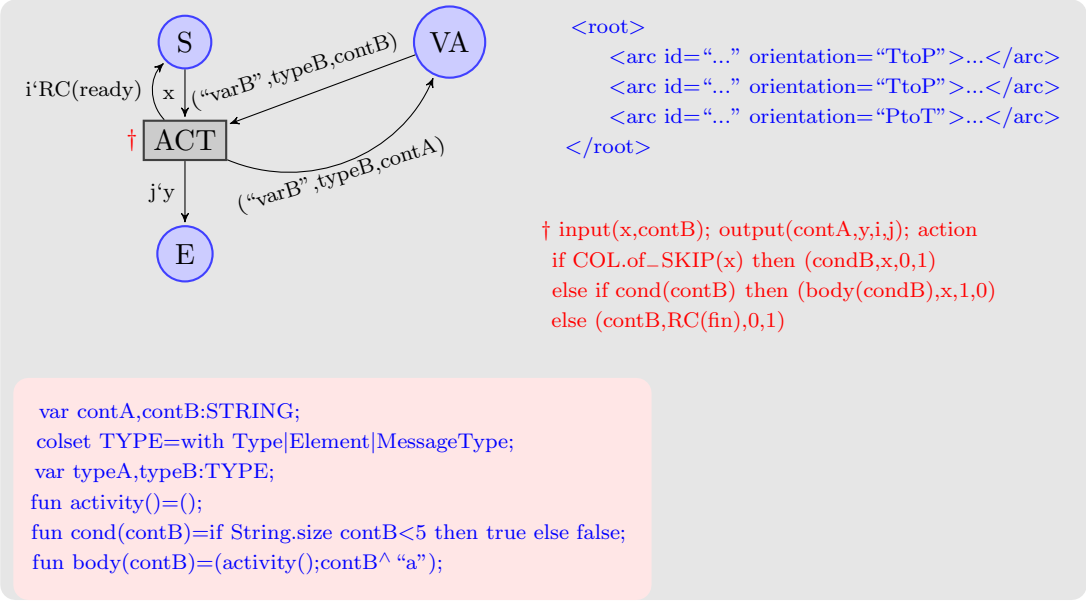


Figure 6.18: The template for while activity.

Template for While Activity

In this section, the template for while activity is proposed. As compared to the multi-way conditional branching in switch, while offers a two-way branching with looping.

Figure 6.18 illustrates the template for while activity. The template consists of three arcs that needs to be added to the template for a basic activity.

As with any basic activity, a *ready* token in place S signifies that the activity is ready to execute. The transition ACT removes this token and uses the function *cond()* to determine the outcome of condition. Usually this function requires the content of a variable that is fetched using an arc from place VA. If the condition is satisfied, the function *body()* is called to execute the underlying activities that are specified in function *activity()*. Furthermore, in order to prevent infinite looping, this function also manipulates the content of variable that is used by function *cond()*. Thereafter the token *ready* is sent back to place S in order to begin the next cycle of looping.

Alternatively, if the condition is not satisfied, the underlying activities are not executed (i.e. *body()* is skipped) and a *fin* token is sent to E. By default, the function *cond()* checks if the content of a variable has less than 5 characters. Consequently, in order to prevent an infinite loop, the function *body()* adds a character to it. These functions are customized based on the object model for the BPEL specification.

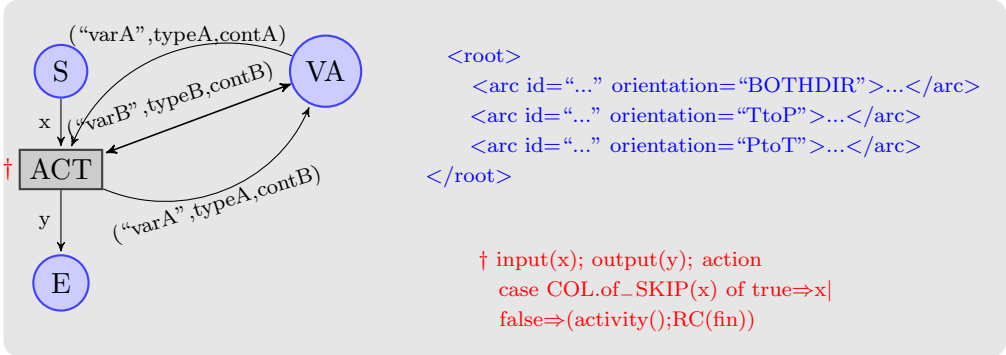


Figure 6.19: The template for assign activity.

Template for Assign Activities

In this section, the template for assign is proposed. It is used to copy data between variables and expressions. The template consists of arcs that are used to copy the contents of a variable into other. This template is used with the template for a basic activity to obtain the model for assign activity.

Figure 6.19 illustrates the template for assign activity. BPEL assign activity requires specifying a source (using $< from >$) amd a destination (using $< to >$) for each copy operation (specified by $< copy >$). In order to ensure that the source variable is not modified, it is accessed in Figure 6.19 using a bi-directional arc. The other two arcs fetch the token for destination variable and replace it with an updated content.

Template for Handler Activities

In this section, the template for handler activities are proposed. As discussed earlier, each scope can have its own set of handlers.

Figure 6.20 illustrate the model for event and fault handler activities. The XML template is similar to that of flow and sequence and therefore not shown. Considering that an event might have an associated time delay, event-handlers are represented using timed CPN. For instance a token

1‘onAlarm(1)@IntInf.fromInt 20

has a time stamp of 20. Consequently it will wait until all tokens with a lower time stamp enable a transition. The time stamp must be of type *Time.time* and not an integer. Therefore

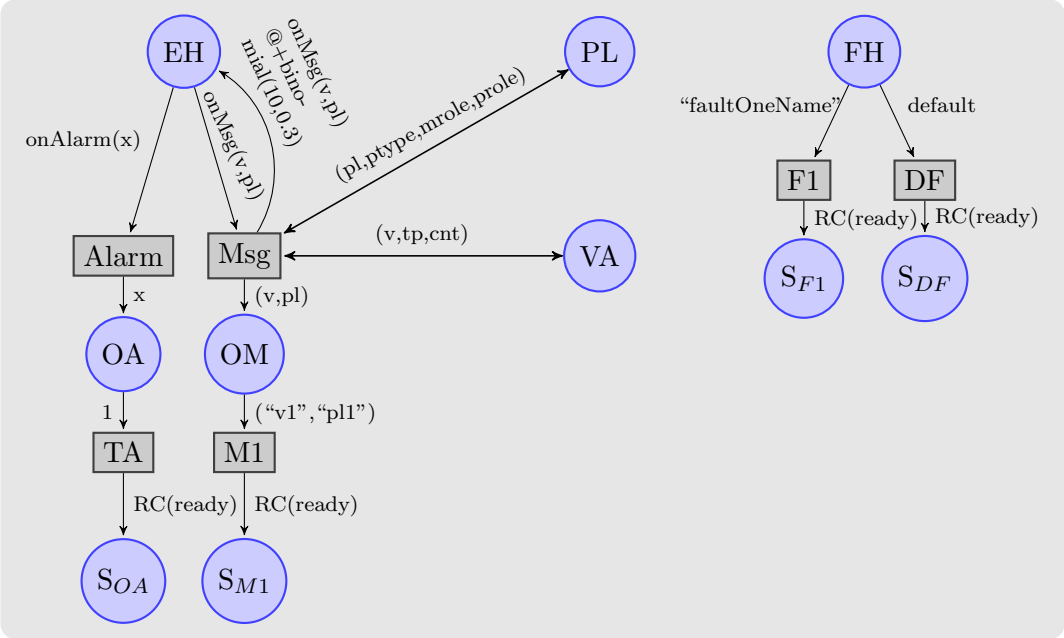


Figure 6.20: The template for handler activities.

an operation *IntInf.fromInt* is used to convert an integer into *Time.time* type. The colour-set for place EA is defined as

- colset ONMESSAGE=product STRING*STRING timed;
- colset ONALARM=INT timed;
- colset EHandlers=union OnMessage:ONMESSAGE+OnAlarm:ONALARM timed;

While an ONALARM token is used to model a time-out, an ONMESSAGE token offers an operation and waits for its invocation. The instant at which a time-out or operation occur depend on the time-stamp attached to these tokens. In either case, an appropriate underlying activity is triggered by adding a *ready* token into its start place. However, unlike time-outs (that can occur only once), an operation offered can be invoked any number of times. Consequently the transition *Msg* in Figure 6.20 replace an ONMESSAGE token after removing it from *EH*. In order to schedule the next operation invocation, a time-delay is added to the time stamp. CPN tools offer various *random distribution functions* that can be used to calculate this delay. Table 6.5 lists some of these functions.

Each ONALARM token is assigned a unique number in order to identify the source of a time-out. This value is bound to variable 'x' for transition *Alarm* executes and copied

Table 6.5: The random distribution functions offered by CPN tools

Function	Conditions	Mean	Variance
bernoulli(p:real) : int	0.0<=p<=1.0	p	p(1-p)
binomial(n:int, p:real) : int	n>=1, 0.0<=p<=1.0	np	np(1-p)
chisq(n:int) : real	n>=1	n	2n
discrete(a:int, b:int) : int	a<=b	(a+b)/2	((b-a+1)^2-1)/12
erlang(n:int, r:real) : real	n>=1, r>0.0	n/r	n/r^2
exponential(r:real) : real	r>0.0	1/r	1/r^2
normal(n:real, v:real) : real	-	n	v
poisson(m:real) : int	m>0.0	m	m
student(n:int) : real	n>=1	0	1/n-2
uniform(a:real, b:real) : real	a<=b	(a+b)/2	((b-a)^2)/12

to place *OA*. The place *OA* has an outgoing transition for each time-out that trigger the corresponding underlying activity.

Each ONMESSAGE token has a unique combination of partnerlink and variable names. When the transition *Msg* executes, this combination is copied to place *OM*. Similar to *OA*, *OM* has an outgoing transition for each operation that trigger the corresponding underlying activity.

In case of fault-handlers, each fault is specified using a separate $< catch >$ activity. They are assigned unique ID based on the *faultName* and/or *faultVariable* attributes specified for the corresponding catch activity. The place *FH* has an outgoing transition for each fault that triggers the underlying fault-handling activity.

The template for basic-activity is modified marginally in order to report faults and is shown in Figure 6.21. In addition to the two possible execution sequence discussed earlier (i.e. skipped and normal execution), an activity can also have a faulty execution. In such scenarios, a token containing the fault-ID is sent to the fault-handler place for local scope. Furthermore, no token is added to the place *E* as the fault prevented the activity from "finishing". The token added to *FH* allows the appropriate fault-handler to execute and administer the required corrective action. A trivial fault is often gracefully handled by adding the *fin* token to *E*. However, serious faults might terminate the execution.

6.5 Results

In order to evaluate the proposed framework, we have created a schemas for each of the proposed Coloured Petri nets (CPNs) templates. This was done using the tool *Trang* [tra, 2011],

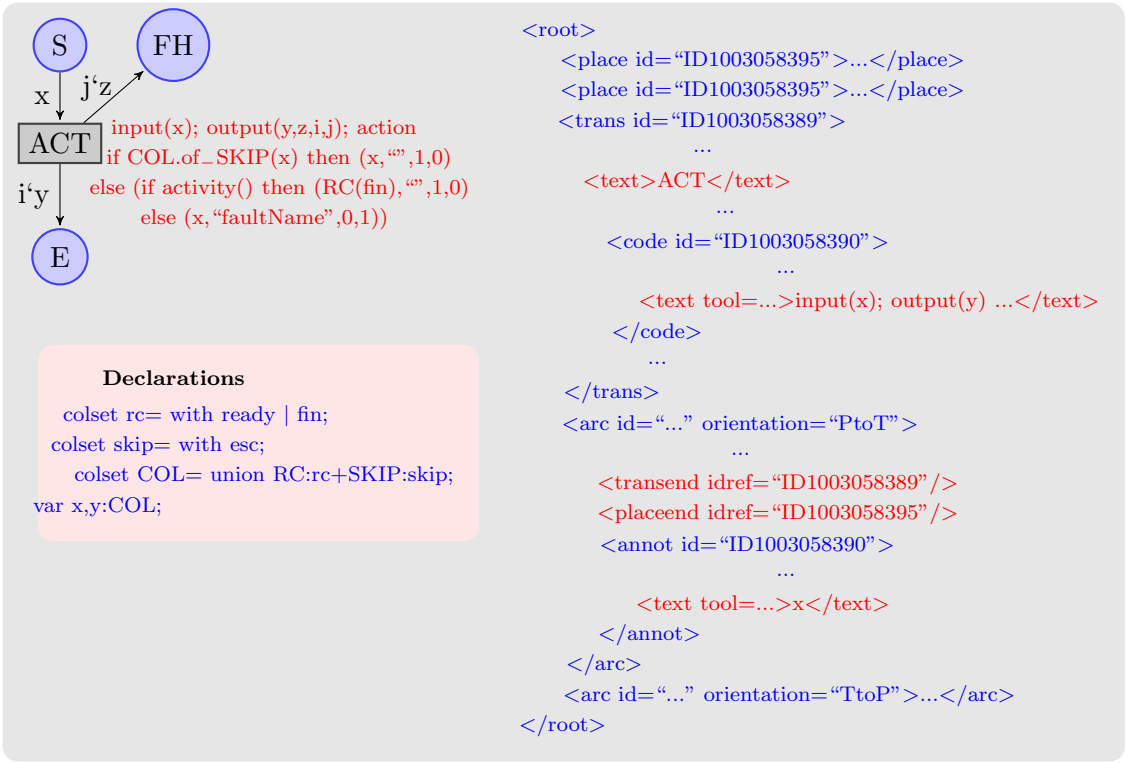


Figure 6.21: The modified template for basic-activity that can report faults.

as explained previously. Each schema define the structure of CPN template corresponding to a BPEL activity. They are used by the JAXB 2 compiler to transform the bean-factory into a formal model.

We administer the experiment and record the results on a desktop with 1.83 GHz Core 2 Duo processor, 2GB RAM and running Windows XP SP2 with JRE 1.6.

6.5.1 Test Cases

The framework is tested on four related BPEL specifications from [Juric, 2006]. They define a simple business process for business travel wherein the client supplies employee name, destination, departure date and return date when invoking the process. Thereafter the process determines the employee travel status by invoking an appropriate web-service. The results from this invocation along with those supplied by client are used to check the price for American and Delta airlines using their exposed web-services. Finally the BPEL process selects the airline offering lower price and returns the itinerary to the client. The required web-services are assumed to exist.

Table 6.6: The difference between BPEL specifications used

M1	**M2 & M3**	**M4**
$< receive >$	$< receive >$	$< receive >$
$< invoke >$	$< scope >$ $\triangleright < invoke >$ $< scope >$	$< flow >$ $\triangleright < links >$ $\triangleright < invoke >$
$< flow >$ $\triangleright < invoke >$ $\triangleright < invoke >$ $< /flow >$ $< switch >$	$< scope >$ $\triangleright < flow >$ $\triangleright\triangleright < invoke >$ $\triangleright\triangleright < invoke >$ $\triangleright < /flow >$ $\triangleright < switch >$ $< /scope >$	$\triangleright < invoke >$ $\triangleright < invoke >$ $\triangleright < switch >$ $< /flow >$
$< invoke >$	$< scope >$ $\triangleright < invoke >$ $< /scope >$	$< invoke >$

Although all the four selected BPEL specifications serve the same purpose, they use different set of activities to achieve the same. Table 6.6 illustrates the difference between these specifications. The first row corresponds to receiving the request from the client. Based on the information attached to this request, the travel information is fetched in the second row. Thereafter the Delta and American airline web-services are invoked and the better offer is determined. Finally the result is returned in the last row.

The specification M1 uses $< flow >$ and $< sequence >$ activities to control the business process work-flow. The specifications M2 and M3 extend M1 by enclosing individual operations in scope. Although M2 and M3 look similar, only M3 has event-handlers for each scope. Considering that event handlers can increase the cost of analysing a model (owing to simultaneous active instances), both M2 and M3 are investigated to determine the additional costs. The specification M4 uses links for the synchronization of its activities.

6.5.2 Empirical Results

Figure 6.22 illustrates the total number of activities in each of the test-cases. The specification M3 and M4 are found to have the maximum and minimum number of activities. As pointed out previously, the event handlers account for the additional activities in M3 as compared to M2.

Figure 6.23 illustrates the time taken by the framework in transforming these BPEL specifications into CPN models. Except for M4, the transformation time increases with an increase in number of activities. However, despite having the least number of activities, the specification M4 defies the trend in having the maximum transformation time.

Figure 6.24 illustrates the number of places and transitions (or nodes) in the transformed

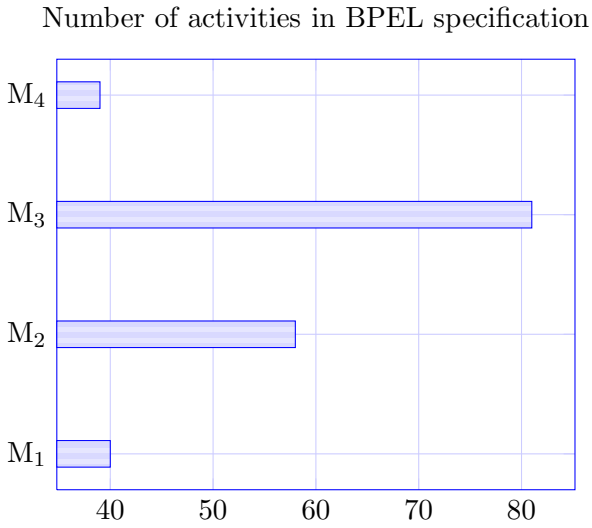


Figure 6.22: Number of activities in BPEL specification.

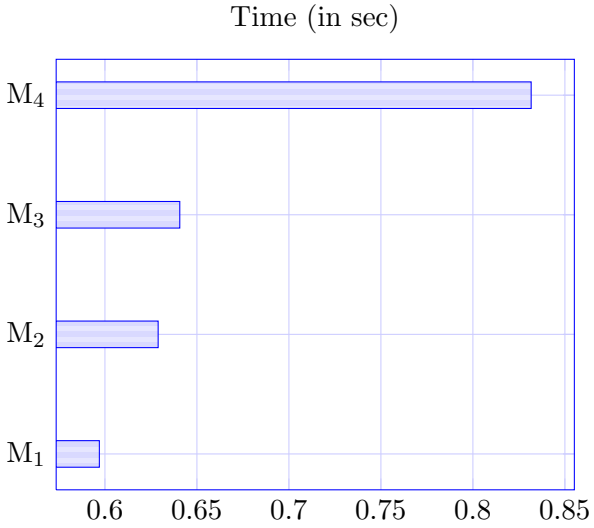


Figure 6.23: The time taken for BPEL to CPN transformation.

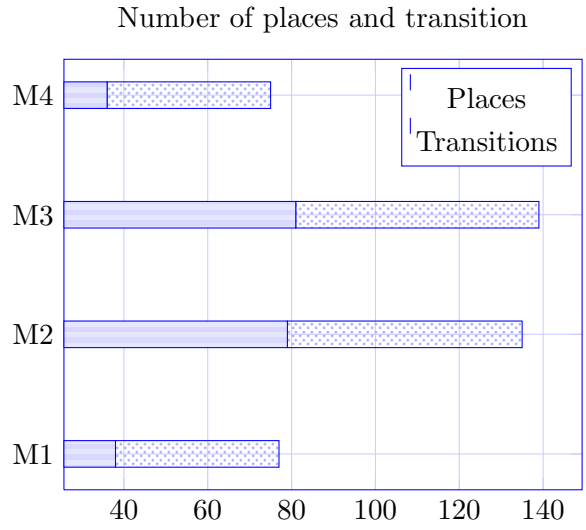


Figure 6.24: Number of places and transition in the model rendered.

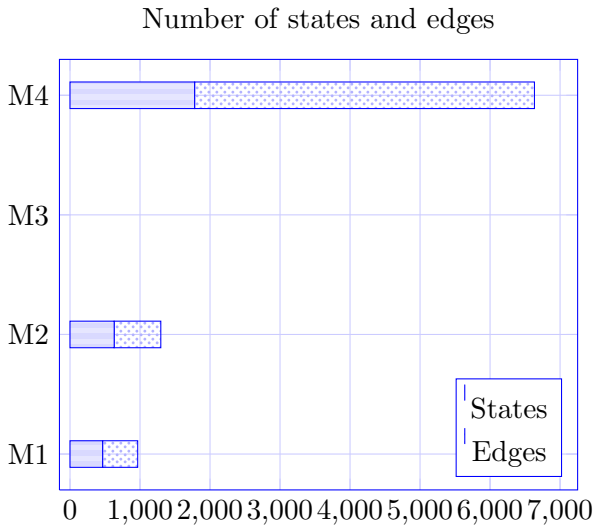


Figure 6.25: Number of states and edges in the state space of transformed models.

models. Barring M4, a higher transformation time is found to produce more nodes and arcs. The model for M4 has the least number of nodes in spite of having the largest transformation time. However, a larger specification (i.e. more activities) is always found to map into a bigger model (i.e. more nodes and arcs).

Figure 6.25 illustrates the number of states (or markings) and edges in the state space for each of the transformed models. In spite of having the minimum number of elements, the

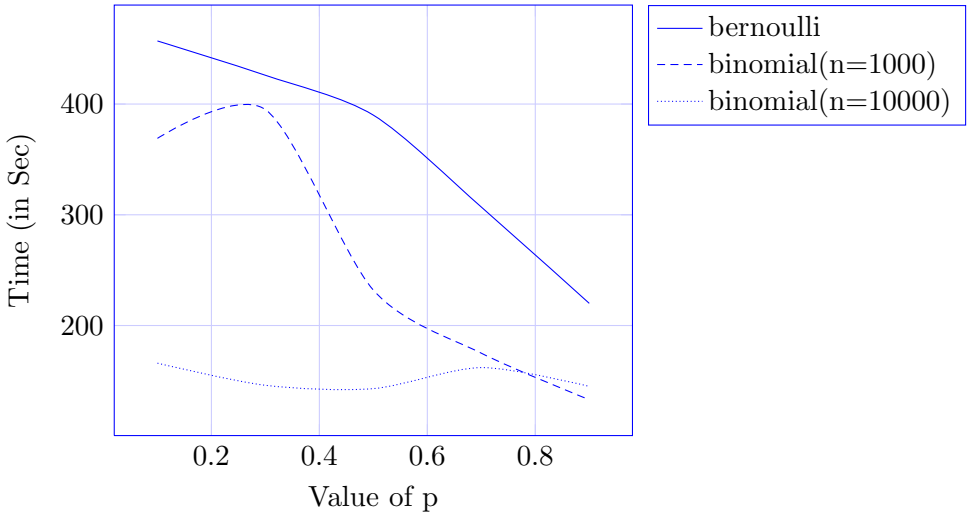


Figure 6.26: The time taken for generating first 50,000 states.

CPN model for M4 is found to produce the highest number of markings. The state-space for M1 and M2 depend on the size of their models. As pointed out previously, the state-space for M3 is infinite owing to the presence of event-handlers. Consequently its entries are missing in Figure 6.25.

However, in order to evaluate M3, the time taken in generating the first 50000 states is plotted in Figure 6.26. The plots in Figure 6.26 corresponds to cases wherein the inter-event time follow a *Bernoulli* or *Binomial* distribution. In the latter case, the plots for n=1000 and n=10000 are manifested. Bernoulli's distribution can also be regarded as a Binomial distribution with n=1.

Figure 6.26 illustrates that the time required to generate first 50,000 states decrease with an increase in value of n. It also decreases as the probability of success increases. However, the plots for n=10000 & n=1000 are found to form a crest for specific probability values.

6.6 Discussion

The empirical results endorse the implemented framework based on the minuscule time taken by it to transform the realistic test-cases considered. The results also underline its importance in automated state-space analysis of BPEL specifications.

Considering that the proposed transformation essentially replaces each BPEL activity with its corresponding CPN model (defined by schema), the transformation time and the

size of rendered model are expected to depend on the size of specification. Barring M4, this is reflected in Figures 6.22, 6.23 and 6.24. The anomalous behaviour of M4 is explained in following paragraph.

As discussed previously, the framework transforms a BPEL activity into its model based on the furnished schemas. Usually a schema is supplied per BPEL activity. When transforming a specification, these schemas are used sequentially in the same order as the corresponding activities appear in the specification. However, the schema for *link* do not actually produce a model. Considering that *links* are applicable for any activity, its schema requires assigning specific values to the model for corresponding activity. The additional time can be attributed to the delay in initialising the appropriate classes with these values.

Figure 6.25 illustrates the size of state space for each of these models. The possible combinations of status value for links and the path of execution for each of these combinations contribute towards the large state-space of M4. As pointed out previously, M3 has an infinite state space owing to the presence of event-handlers. Therefore it is not plotted in Figure 6.25.

However, in order to evaluate M3, the time taken in generating first 50,000 unique states is plotted against various random distribution function that determine the frequency of events occurring. When events occur frequently (small n, small p), it takes long to generate the 50,000 states. This is because repeated execution of the part of the model corresponding to an event does not change the state. The state changes only on the execution of main flow.

6.7 Summary

This chapter contributes towards enhancing the safety and reliability of a SOA based application by verifying its underlying service composition. We identified the issues with BPEL, the de-facto language for service composition, and proposed techniques to formalize its activities in order to model, simulate and verify a BPEL specification. The transformation is fully automatic and renders a CPN model. The main advantages of our approach are:

- The object model proposed helps in determining the relationship among BPEL activities. It forms a basis for the proposed templates and acts as the antecedent of the CPN model obtained after transformation.

- Unlike related techniques that simply propose models for each BPEL activity, our technique propose templates of varying granularity based on the relationship among BPEL activities.

- The proposed templates for BPEL activities are feature complete and confirm to the DTD specified for CPN tools.

The extensive tool support for coloured Petri Nets further enhance the significance of our technique.

Chapter 7

Conclusion

This thesis addresses the intriguing issues related to model-checking and verification of BPEL specifications. The research has been motivated by the necessity for a wider use of formal methods in enhancing the safety and reliability of software systems. Furthermore the research has also been driven by the lack of robustness in loosely coupled SOA based applications.

As observed previously, model-checking is a rigorous technique wherein all possible behaviours of the system are scrutinised exhaustively to determine a problem. Consequently model-checking is expected to identify all issues in a system. However, there are significant time and memory requirements in model-checking a system. Therefore the scope of model-checking has hitherto been limited to critical systems where reliability is excessively important. Nevertheless, with our ever increasing dependence on software in everyday life (e.g. traffic signals, elevators), skipping model-checking amounts to risking millions of human lives.

Chapters 3 and 4 of this thesis propose novel techniques to reduce the time and memory requirements for model-checking. The envisioned reduction in memory requirements for model-checking is realised by storing states as the difference from previous or nearest state. The time reduction is attributed to generating the reachability graph for each module of a hierarchical model in parallel. The proposed techniques offer better results for larger models that correspond to contemporary software systems. They also have lower time overhead in reducing the memory requirements. Consequently our techniques would allow model-checking to acquire a bigger role in verification of a wide range of software.

Chapter 5 introduces a technique to install hierarchy into into a flat model. Considering that a hierarchical model is often exponentially more succinct as compared to its equivalent

flat model, they are easier to analyse and maintain. Furthermore this allows the time reduction techniques from Chapter 4 to be applied for flat models after installing hierarchy. When compared to the existing techniques, the proposed method renders an equivalent succinct model. This ensures the consistency of any analysis technique.

Finally Chapter 6 proposes a framework to model check a BPEL specification. Unfortunately the safety and reliability of SOA based systems entirely depend on the precision of service descriptions. Consequently any implicit assumption or unforeseen usage scenarios can lead to undesirable forms of interactions, such as a deadlock or race condition [Sloan and Khoshgoftaar, 2009]. This is further exacerbated by dynamic service composition wherein services could be added, removed or updated at runtime. The proposed framework transforms a BPEL specification into a hierarchy of DTOs that act as a generic intermediate before the actual formalisation. Considering the ad-hoc nature of existing solutions, the proposed framework offers significant flexibility in formalising a BPEL specification. The framework is open and can be extended for rendering the formal model. This has been demonstrated by formalising the intermediate java-beans into an XML based formal model.

7.1 Results

This section highlights our results in extending and improving the existing solutions and answering the research questions posted in Section 1.3.

Memory efficient state-space analysis technique

A new storage technique is proposed to reduce the memory costs otherwise involved in model-checking service compositions. The proposed technique requires storing states as the difference from one of its neighbouring states. Such a setup offers several advantages over related techniques:

- The technique is generic and applicable for any modeling language. Consequently it outweighs earlier solutions, such as [Evangelista and Pradat-Peyre, 2005] that are only applicable for Petri net formalisms (and its extensions).

- It is based on exhaustive storage technique wherein each distinct state of the system is stored to identify and purge the duplicate states. However, solutions based on partial storage techniques [Christensen et al., 2001] often fail to identify the duplicate states.

- The difference algorithm for compressing a state is reversible and allows reinstating a condensed state. Consequently false negative and false positive situations can be prevented by expanding the stored states before comparison.
- The proposed technique provides a 95% reduction in memory requirements with only twice the processing time. Other solutions, such as [Schmidt, 2003; Evangelista and Pradat-Peyre, 2005; Holzmann, 1997] have considerably large processing times and offer significantly less memory reduction.
- The technique is flexible in allowing a choice of neighbouring states to be used for calculating the difference form of a state.
- The proposed technique performs better for contemporary systems that have relatively high levels of complexity.

Time efficient state-space analysis technique

A novel method is proposed to reduce the time requirements for model-checking a service composition. The solution necessitates the composition to be formalised as a hierarchical model and the associated reduction is attributed to the concurrent exploration of its modules. The outcome of this exploration is stored using special data-structures that acts as a repository of corresponding module behaviour. A module can use these data-structures to determine the behaviour of any other module without actually executing it. The proposed technique offers several advantages over other related solutions:

- The proposed solution only necessitates a hierarchical model. This is less stringent than earlier solutions, such as [Evangelista and Pradat-Peyre, 2006; Elgaard, 2002] that necessitate the presence of stubborn sets or symmetry in the model.
- Contrary to the proposed technique, the solutions based on stubborn sets and symmetry are NP-hard and use heuristic estimations [Varpaaniemi, 2000; Clarke et al., 1998].
- The technique is applicable for all modeling languages that define a notion of hierarchy.
- The proposed technique offers 86% reduction in delay for generating the first 25,000 states. This significantly outweighs the reduction offered by related solutions [Evangelista and Pradat-Peyre, 2006; Kristensen and Valmari, 1998].

A technique for reducing the size of a model exponentially

The proposed technique renders an exponentially more succinct representation of a service composition by embracing the notion of hierarchy. A hierarchical model consists of a set of modules wherein each module represents a system component. In such a setup, the module for a high-level component refers to its underlying components using their module names or reference. This avoids the explosion in including the actual representation of underlying components. The proposed technique outperforms the earlier solutions in several aspects:

- The proposed technique installs hierarchy to render an equivalent succinct model. However, the reduced model obtained using other techniques, such as [Berthelot, 1986; Haddad, 1990; Evangelista et al., 2005], is not equivalent to the original model. Furthermore, these techniques fail to preserve the properties of the original net, other than those specifically targeted.

- The proposed technique aims to increase the analysability and maintainability of formal models, while other techniques, such as [Berthelot, 1986; Haddad, 1990; Evangelista et al., 2005], only target diminishing the state-space by reducing the number of execution traces to be analysed.

- While other techniques reduce the size of a model by merging its elements (e.g. merging two or more places or transitions in Petri nets) based on certain conditions, the proposed technique does the same by installing hierarchy. The hierarchical models help in identifying the overall architecture of the system, understanding its dependencies, visualising the flow of information through it, identifying its capabilities and limitations and calculating its complexity [Christopher, 2003]

- The proposed technique requires finding structural similarity prior to installing hierarchy. Consequently it is limited to graph based formal models (e.g. Petri Nets).

A technique for modeling, simulating and verifying a BPEL specification

A verification framework is proposed to formalise a BPEL specification by transforming it into an XML based formal model. This is done by extending the *Spring framework* to represent each BPEL activity using a *Java bean*. The framework instantiates the beans corresponding to activities in a BPEL specification and injects the dependencies to yield a *bean-factory*. Thereafter *Java Architecture for XML Binding (JAXB) 2* APIs are used to transform the

bean-factory into an XML based formal-model (e.g. Coloured Petri nets (CPN) [Jensen and Kristensen, 2009]) or an interchange format (e.g. Petri Net Markup Language (PNML)) for simulation and verification. Our technique offers several advantages over other related solutions:

- The existing techniques [Foster et al., 2003; Kang et al., 2007; Fu et al., 2004] are ad-hoc and temporary in targeting specific modeling languages. The proposed framework targets all modeling languages using a generic intermediate specification.
- Contrary to most of the existing solutions [Foster et al., 2003; Kang et al., 2007], the proposed solution allows automatic transformation.
- The proposed framework uses Data Transfor Objects (DTOs) as intermediates. DTOs are commonly used design pattern in software engineering for storing and transferring data [Crawford and Kaplan, 2003].
- The proposed framework allows plugging a component to transform intermediate DTOs into a formal model.
- A Java Architecture for XML Binding (JAXB) 2 APIs based component has been proposed to demonstrate the transformation of DTOs into an XML based formal model.
- The framework uses Spring framework and thereby offers all its advantages (e.g. loosely coupled, lightweight etc.).
- An object model has been proposed to identify the hierarchical relationship among BPEL activities. This helps in mapping the BPEL activities into Java Beans.
- The transformation time is significantly low (less than .7 sec).

7.2 Discussion

This section discusses the proposed solutions in regards to the statement of the problem presented in Section 1.2. The solutions are also evaluated based on their ability to address the issues.

The reliability and robustness of a service composition is enhanced by proposing a verification framework in Chapter 6. As discussed earlier using Figure 1.3, a SOA based application consists of a hierarchy of services and a failure at any level could break the application. However, the service level agreement (SLA) with the vendors providing these web-services assure

their reliability and quality of service (QoS). Consequently the vulnerabilities in a SOA based application are introduced on composing these services to create the application. In order to enhance the reliability and correctness of a SOA based system, the composition must be exhaustively verified for any single point of failure (SPOF). Considering that business process execution language (BPEL) is the de-facto industry standard for web-service composition, this essentially involves verifying a BPEL specification. The proposed framework verifies a composition by formalising the corresponding BPEL specification before verifying it using a model-checking tool. This in turn allows identifying the SPOFs in a composition and rectifying them.

BPEL has emerged out of Web Services Flow Language (WSFL) [IBM, 2001] of IBM and XLANG [Thatte, 2001] of Microsoft. However WSFL is a graph based language while XLANG is a block based language. Consequently BPEL has both block (e.g. sequence) and graph (e.g. flow) based elements. The contrasting concepts in the base languages have caused many inconsistencies in the BPEL language that could undermine a service composition [Wohed et al., 2002]. Furthermore the textual specification of BPEL and its lack of mathematical semantics prevent formal methods to be directly applied to a BPEL specification [van der Aalst, 2003; Schmidt and Stahl, 2004].

Most of the existing solutions directly formalise a BPEL specifications using a specific modeling language before verifying it [Kang et al., 2007; Stahl, 2005]. Despite these being legitimate solutions, they are only applicable for the targeted modeling language. Furthermore they require scanning a BPEL specification manually and replacing each activity with its proposed formal-model. Apart from being a cumbersome process, such an exercise is error-prone and time-consuming. The proposed method 1) automates the transformation, and 2) is applicable for a range of modeling languages. In this pursuit, it transforms the BPEL specification into intermediate data transfer objects (DTOs) before the actual formalisation. DTOs are generic intermediates that could be programmatically accessed and transformed into a range of formal models. Considering that DTOs are commonly used design pattern in software engineering for storing and transferring data [Crawford and Kaplan, 2003], they are used as generic intermediate specifications in the proposed solution.

The aforementioned solution is made more appealing by reducing the time and memory requirements for model-checking. As discussed earlier in Section 1.2, model-checking a contemporary software system has significant overheads owing to the huge state-space of the latter. The time overhead is attributed to the analysis of entire state-space for a set of undesirable properties. Furthermore, considering that some systems repeatedly reach one or

more states during execution, model checker remembers the analysed states by storing them in memory. Such state of affairs account for the memory overhead.

Chapter 3 proposes a technique to reduce the memory requirements by storing the states as the difference from an adjoining state. However, since this difference accounts for a *change in state* rather than the state itself, it is possible that 1) two dissimilar states have the same difference state, and 2) two similar states have different difference states. The latter is possible because the similar states could have different adjoining states that are used for calculating the difference. Such state of affairs could lead to false positive and false negative scenarios. A false positive scenario arises when the two dissimilar states have the same difference state. Similarly a false negative scenario arises when two similar states have different difference states. The proposed method prevents false positive and false negative scenarios by devising a method to regenerate the explicit form for a difference state. Thereupon the states are expanded before comparison to rule out any ambiguities. Asserting that the change in state is always smaller than the state itself, this technique offers up to 95% reduction in memory requirements. The solution is based on the exhaustive storage technique discussed in Section 3.3. Furthermore it performs better for contemporary systems that have relatively high levels of complexity.

Chapter 4 proposes a novel method to reduce the time requirements for model-checking a service composition. The method is applicable for hierarchical models and the associated reduction is obtained by concurrently exploring the modules of a hierarchical model. The outcome of exploring a module is stored in special data-structures that act as the repository of corresponding module behaviour. Thereupon a module can use these data-structures to determine the behaviour of another module without actually executing it. The method offers up to 86% reduction in time requirements.

The proposed techniques also allows a human modeler to analyse and determine the SPOFs in a composition. In this pursuit, Chapter 5 proposes a method for installing hierarchy into the formal representation of a service composition. A hierarchical model offers different levels of abstraction and expressiveness. Consequently, regardless of the size of the model for a service composition, a human modeler can analyse and determine the SPOFs. The hierarchy is installed by identifying the structurally similar components in a flat model and constituting a module for each of them.

Nevertheless, the advantages offered by the proposed solutions have some limitations. Primarily the proposed verification technique cannot automatically formalise a BPEL specification into every existing modeling language. This is essentially because certain formal

representations cannot be programmatically generated (e.g. process algebras). However such limitations can be easily circumvented by formalising the specification into an intermediate formal model before obtaining the required representation. The intermediate formal model must be 1) easy to obtain from the intermediate DTOs, and 2) easy to transform into the targeted model.

Considering the additional processing time in 1) calculating the difference form for the generated states, 2) storing and retrieving the states, and 3) regenerating explicit states before comparison, the proposed memory reduction technique has a time overhead. Experimental results indicate that the processing time for model-checking is doubled when using the proposed solution.

In addition, the time reduction techniques cannot be applied for non-hierarchical models. Although this limitation is addressed by the method for installing hierarchy into a flat model, this is only applicable when the modeling language used has the semantics of hierarchy and structural similarity.

7.3 Future Work

This section outlines the course of future research in further enhancing the proposed solutions.

Memory requirements for Model-Checking

The proposed technique reduces the memory requirements by storing a state as the difference from its nearest or immediately previous state. Although the results exemplify the prowess of our method, it has an associated time overhead. This overhead in combination with the processing delay could counter the memory reduction achieved.

Considering the significant strides in parallel and distributed computing techniques, their use in the proposed method should further reduce the time overhead. The availability of multiple computers in a distributed technique leads to 1) an increase in memory available for model-checking and 2) a decreases in delay (with an increase in processing resources). However, depending on the network latency, there could be an additional time overhead in distributed model-checking. Consequently the model-checking algorithm should be optimised to require minimum network communications. Such optimisations could involve piggybacking, compression etc.

Time requirements for Model-Checking

The proposed technique for reducing the model-checking time is based on modular state-space generation. Such techniques generate the reachability graph for each module independently before composing them to generate the system reachability graph. However, this technique has hitherto been sparingly used. This is essentially because modular techniques are not applicable for flat models.

Considering the exquisite results obtained for the proposed time-reduction method, modular techniques should be further researched to enhance their applicability. This could be done by introducing auto formalisation tools targeting hierarchical models instead of flat models. A hierarchical model could be generated by identifying the identical components of a system. In such a setup each component of the system could be represented by a module and they would together constitute the hierarchical model. This would also enhance the analysability and maintainability of the models.

Installing Hierarchy into Models

The proposed technique for installing hierarchy is based on identifying the structural similarities in a formal model. However, it is only applicable for graph based models (e.g. Petri nets and its extensions, automata) that define the notion of structural similarity. Consequently the proposed technique cannot be applied for certain modeling languages (e.g. Promela [pro, 2011], SMV [McMillan, 2000]).

Considering the obvious advantages in installing hierarchy into a flat model (i.e. they can be easily analysed and maintained, they can be subjected to the proposed modular technique in chapter 4 etc.), the proposed technique should be extended for other modeling languages. This could be done by using alternative techniques in identifying the similar components of a model. Such techniques could include semantic similarity, logical similarity etc [Maguitman et al., 2005].

Formalising a BPEL Specification

The proposed framework transforms a BPEL specification into a hierarchy of DTOs before formalising them into an XML based formal model. The use of DTOs in software engineering as design patterns for storing and transferring data [Crawford and Kaplan, 2003] legitimises their role as generic intermediates. However the existence of a significant number of non-XML based modeling languages (e.g. Promela [pro, 2011], SMV [McMillan, 2000]) necessitates the

transformation of DTOs into these languages. Consequently the proposed framework should be extended with additional automatic transformations for DTOs.

This could be done by mapping BPEL activities into the alternate modeling language being used. This mapping can thereupon be used for programmatically formalising the DTOs into the target formal language. This transformation might require certain degree of manual interaction depending on the target modeling language. However if this transformation is excessively difficult, the DTOs could be transformed into an intermediate modeling language before transforming it into the target language.

Bibliography

Eclipse Modeling Framework (EMF), July 2010. URL http://www.eclipse.org/modeling/emf.

Promela Manual, January 2011. URL http://spinroot.com/spin/Man/promela.html.

Basic Spin Manual, June 2007. URL http://spinroot.com/spin/Man/Manual.html.

Spring Android, January 2011a. URL http://www.springsource.org/spring-android.

Spring Mobile, January 2011b. URL http://www.springsource.org/spring-mobile.

Spring.Net Application Framework, January 2011c. URL http://www.springframework.net/.

Trang Manual, January 2011. URL http://www.thaiopensource.com/relaxng/trang.html.

R. Alur. Formal analysis of hierarchical state machines. In *Verification: Theory and Practice*, volume 2772 of *Lecture Notes in Computer Science*, pages 434–435. Springer Berlin / Heidelberg, 2004. ISBN 978-3-540-21002-3.

R. Alur and M. Yannakakis. Model checking of hierarchical state machines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):273–303, 2001.

P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 1 edition, 2008. ISBN 0521880386, 9780521880381.

T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. *Business Process Execution Language for Web Services Version 1.1*, May 2003.

J. Arias-Fisteus, L. S. Fernández, and C. D. Kloos. Formal verification of BPEL4WS business collaborations. In *E-Commerce and Web Technologies*, pages 76–85, 2004.

A. Arkin, S. Askary, B. Bloch, F. Curbera, Y. Goland, N. Kartha, C. K. Liu, V. Mehta, S. Thatte, P. Yendluri, A. Yiu, and A. Alves. *Web Services Business Process Execution Language Version 2.0*, December 2005.

A. A. Bayazit and S. Malik. Complementary use of runtime validation and model checking. In *Proceedings of the 2005 IEEE/ACM International conference on Computer-aided design*, ICCAD '05, pages 1052–1059, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7803-9254-X. URL http://portal.acm.org/citation.cfm?id=1129601.1129750.

B. Beizer. *Software testing techniques*. Van Nostrand Reinhold Co., 1990. ISBN 0-442-20672-0.

G. Berthelot. Checking properties of nets using transformation. In *Advances in Petri Nets 1985, covers the 6th European Workshop on Applications and Theory in Petri Nets-selected papers*, pages 19–40, London, UK, 1986. Springer-Verlag. ISBN 3-540-16480-4. URL http://portal.acm.org/citation.cfm?id=647733.735159.

D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker blast: Applications to software engineering. *International Journal on Software Tools for Technology Transfer (STTT)*, 9:505–525, October 2007. ISSN 1433-2779. doi: 10.1007/s10009-007-0044-z. URL http://portal.acm.org/citation.cfm?id=1296691.1296695.

J. Billington, S. Christensen, K. Van Hee, E. Kindler, O. Kummer, L. Petrucci, R. Post, C. Stehno, and M. Weber. The petri net markup language: concepts, technology, and tools. In *Proceedings of the 24th international conference on Applications and theory of Petri nets*, ICATPN'03, pages 483–505, Berlin, Heidelberg, 2003. Springer-Verlag. ISBN 3-540-40334-5. URL http://portal.acm.org/citation.cfm?id=1760066.1760101.

J. Bondy and U. Murty. *Graph Theory*. Springer Publishing Company, Incorporated, 2008. ISBN 978-1-84996-690-0.

P. Bonet, C. Lladó, R. Puigjaner, and W. J. Knottenbelt. PIPE v2.5: A Petri Net Tool for Performance Modelling. In *23rd Latin American Conference on Informatics (CLEI 2007)*, September 2007. URL http://pubs.doc.ic.ac.uk/pipe-clei/.

I. N. Bronshtein, K. A. Semendyayev, and K. A. Kirsch. *Handbook of mathematics (3rd ed.)*. Springer-Verlag, London, UK, 1997. ISBN 3-540-62130-X.

F. Casati, S. Ilnicki, L.-J. Jin, V. Krishnamoorthy, and M.-C. Shan. eflow: A platform for developing and managing composite e-services. In *Proceedings of the Academia/Industry Working Conference on Research Challenges*, AIWORC '00, pages 341–, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0628-3. URL http://portal.acm.org/citation.cfm?id=786016.786038.

D. Chakraborty, F. Perich, A. Joshi, T. W. Finin, and Y. Yesha. A reactive service composition architecture for pervasive computing environments. In *PWC '02: Proceedings of the IFIP TC6/WG6.8 Working Conference on Personal Wireless Communications*, pages 53–62, 2002.

P. P.-W. Chan and M. R. Lyu. Dynamic web service composition: A new approach in building reliable web service. In *Advanced Information Networking and Applications*, pages 20–25, 2008. doi: 10.1109/AINA.2008.133.

J. Chen and H. Cui. Translation from adapted uml to promela for corba-based applications. In *11th International SPIN Workshop*, pages 234–251, 2004.

S. Christensen and L. Petrucci. Modular state space analysis of coloured petri nets. In *Proceedings of the 16th International Conference on Application and Theory of Petri Nets*, pages 201–217. Springer-Verlag, 1995.

S. Christensen, L. M. Kristensen, and T. Mailund. A sweep-line method for state space exploration. In *TACAS 2001: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 450–464, 2001.

G. Christopher. Software modeling introduction. *Borland White Paper*, March 2003. URL http://www.borland.com/resources/en/pdf/white_papers/software_modeling_introduction.pdf.

E. Clarke, D. Long, and K. McMillan. Compositional model checking. In *Proceedings of the Fourth Annual Symposium on Logic in computer science*, pages 353–362, Piscataway, NJ, USA, 1989. IEEE Press. ISBN 0-8186-1954-6.

E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000. ISBN 978-0-262-03270-4.

E. M. Clarke and S. Berezin. Model checking: Historical perspective and example (extended abstract). In *Proceedings of the International Conference on Automated Reasoning with*

Analytic Tableaux and Related Methods, TABLEAUX '98, pages 18–24, London, UK, UK, 1998. Springer-Verlag. ISBN 3-540-64406-7. URL http://portal.acm.org/citation.cfm?id=646889.756979.

E. M. Clarke, E. A. Emerson, S. Jha, and A. P. Sistla. Symmetry reductions in model checking. In *Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 147–158. Springer Berlin / Heidelberg, 1998. ISBN 978-3-540-64608-2. doi: 10.1007/BFb0028741.

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001. ISBN 0262531968.

W. Crawford and J. Kaplan. *J2EE Design Patterns*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2003. ISBN 0596004273.

F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, and S. Weerawarana. *Business Process Execution Language for Web Services Version 1.0*. IBM,Microsoft,BEA Systems, July 2002.

A. David and M. O. Möller. From HUPPAAL to UPPAAL: A translation from hierarchical timed automata to flat timed automata. Research Series RS-01-11, BRICS, Department of Computer Science, University of Aarhus, Mar. 2001.

L. Elgaard. *The Symmetry Method for Coloured Petri Nets - Theory, Tools and Practical Use*. PhD thesis, University of Aarhus, Denmark, 2002.

E. A. Emerson and A. P. Sistla. Symmetry and model checking. *Formal Methods in System Design*, 9(1-2):105–131, 1996. ISSN 0925-9856. doi: http://dx.doi.org/10.1007/BF00625970.

S. Evangelista. High level petri nets analysis with helena. In *Lecture Notes in Computer Science: Applications and Theory of Petri Nets 2005: 26th International Conference, ICATPN 2005, Miami, USA, June 20-25, 2005. / Gianfranco Ciardo, Philippe Darondeau (Eds.)*, volume 3536, pages 455–464. Springer Verlag, June 2005.

S. Evangelista and J.-F. Pradat-Peyre. Memory efficient state space storage in explicit software model checking. In *Proceedings of the 12th International SPIN Workshop on Model Checking of Software*, volume 3639 of *Lecture Notes in Computer Science*, pages 43–57. Springer-Verlag, 2005.

S. Evangelista and J.-F. Pradat-Peyre. On the computation of stubborn sets of colored petri nets. In *ICATPN '06: Proceedings of the 27th International Conference on Application and Theory of Petri Nets*, pages 146–165, 2006.

S. Evangelista, S. Haddad, and J.-F. Pradat-Peyre. Syntactical Colored Petri Nets Reductions. In *ATVA 2005: Proceedings of the 3rd International Symposium on Automated Technology for Verification and Analysis*, volume 3707 of *LNCS*, pages 202–216. Springer, 2005.

D. Fahland. Complete abstract operational semantics for the web service business process execution language. Informatik-Berichte 190, Humboldt-Universitat zu Berlin, Sept. 2005. URL http://www.informatik.hu-berlin.de/top/download/publications/Fahland2005_hub_tr190.pdf.

D. Fahland, W. Reisig, and D. F. W. Reisig. Asm-based semantics for BPEL: The negative control flow. In *Proceedings of the 12th International Workshop on Abstract State Machines*, pages 131–151, 2005.

R. Fehling. A concept of hierarchical petri nets with building blocks. In *Proceedings of the 12th International Conference on Applications and Theory of Petri Nets*, pages 148–168. Springer-Verlag, 1993.

A. Ferrara. Web services: a process algebra approach. In *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*, pages 242–251, 2004.

H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based verification of web service compositions. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering, 2003.*, pages 152–161, Oct. 2003.

X. Fu, T. Bultan, and J. Su. Analysis of interacting BPEL web services. In *Proceedings of the 13th World Wide Web Conference*, pages 621–630, New York, NY, USA, 2004. ACM. ISBN 1-58113-844-X. doi: http://doi.acm.org/10.1145/988672.988756.

Y. Gan, M. Chechik, S. Nejati, J. Bennett, B. O'Farrell, and J. Waterhouse. Runtime monitoring of web service conversations. In *Proceedings of the 2007 conference of the center for advanced studies on Collaborative research*, CASCON '07, pages 42–57, New York, NY, USA, 2007. ACM. URL http://doi.acm.org/10.1145/1321211.1321217.

J. Geldenhuys and A. Valmari. A nearly memory-optimal data structure for sets and mappings. In *Proceedings of the 10th international SPIN conference on Model checking software*, pages 136–150, Berlin, Heidelberg, 2003. Springer-Verlag. ISBN 3-540-40117-2. URL http://portal.acm.org/citation.cfm?id=1767111.1767120.

P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996. ISBN 3540607617.

P. Godefroid, G. J. Holzmann, and D. Pirottin. State-space caching revisited. In *Proceedings of the Fourth International Workshop on Computer Aided Verification*, pages 178–191, London, UK, 1993. Springer-Verlag. ISBN 3-540-56496-9. URL http://portal.acm.org/citation.cfm?id=647761.735327.

J.-C. Grégoire. State space compression in spin with getss. In *Proceedings of the Second SPIN Workshop*, pages 3–19. American Mathematical Society, 1996.

O. Grumberg and H. Veith, editors. *25 Years of Model Checking - History, Achievements, Perspectives*, volume 5000 of *Lecture Notes in Computer Science*, 2008. Springer. ISBN 978-3-540-69849-4.

G. Gueta, C. Flanagan, E. Yahav, and M. Sagiv. Cartesian partial-order reduction. In *Proceedings of the 14th international SPIN conference on Model checking software*, pages 95–112, 2007.

S. Haddad. A reduction theory for coloured nets. In *Advances in Petri Nets 1989, covers the 9th European Workshop on Applications and Theory in Petri Nets-selected papers*, pages 209–235, London, UK, 1990. Springer-Verlag. ISBN 3-540-52494-0. URL http://portal.acm.org/citation.cfm?id=647735.735457.

A. Hall. Realising the benefits of formal methods. In *Proceedings of the 7th International Conference on Formal Engineering Methods*, pages 1–4, 2005.

Y. Hao and Y. Zhang. Web services discovery based on schema matching. In *Proceedings of the thirtieth Australasian conference on Computer science - Volume 62*, ACSC '07, pages 107–113, Darlinghurst, Australia, Australia, 2007. Australian Computer Society, Inc. ISBN 1-920-68243-0. URL http://portal.acm.org/citation.cfm?id=1273749.1273762.

D. Harel and A. Pnueli. On the development of reactive systems. In *Logics and models of concurrent systems*, pages 477–498, New York, NY, USA, 1985. Springer-Verlag New York, Inc. ISBN 0-387-15181-8.

G. J. Holzmann. State compression in spin: Recursive indexing and compression training runs. In *Proceedings of Third International SPIN Workshop*, 1997.

G. J. Holzmann. *The SPIN Model Checker : Primer and Reference Manual*. Addison-Wesley Professional, September 2003. ISBN 0321228626.

G. J. Holzmann and A. Puri. A minimized automaton representation of reachable states. *Software Tools for Technology Transfer*, 2:270–278, 1999.

J. Hu, C. Guo, H. Wang, and P. Zou. Web services peer-to-peer discovery service for automated web service composition. In *Proceedings of the third international conference on Network and Mobile computing (ICCNMC'05)*, pages 509–518, 2005.

Web Services Flow Language (WSFL 1.0). IBM, May 2001.

D. Jacobs. Distributed computing with bea weblogic server. In *Proceedings of the 2003 CIDR Conference*, 2003.

K. Jensen. *Coloured Petri Nets Basic Concepts, Analysis Methods and Practical Use*, volume Volume 1,2 and 3. Springer-Verlag, 1996. ISBN 3-540-58276-2.

K. Jensen and L. M. Kristensen. *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*. Springer-Verlag, 2009. ISBN 978-3-642-00283-0.

K. Jensen, L. M. Kristensen, and L. Wells. Coloured petri nets and cpn tools for modelling and validation of concurrent systems. *Int. J. Softw. Tools Technol. Transf.*, 9(3):213–254, 2007.

R. Johnson. *Expert One-on-One J2EE Design & Development*. Wrox Press Ltd., Birmingham, UK, UK, 2002. ISBN 1861007841.

M. Juric. *Business Process Execution Language for Web Services*. Packt Publishing, 2006. ISBN 1-904811-81-7.

H. Kang, X. Yang, and S. Yuan. Modeling and verification of web services composition based on cpn. In *Proceedings of the 2007 IFIP International Conference on Network*

and Parallel Computing Workshops, pages 613–617, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2943-7.

J. Köbler, U. Schöning, and J. Torán. *The graph isomorphism problem: its structural complexity*. Birkhauser Verlag, Basel, Switzerland, Switzerland, 1993. ISBN 0-8176-3680-3.

L. M. Kristensen and S. Christensen. Implementing coloured petri nets using a functional programming language. *Higher-Order and Symbolic Computation*, 17(3):207–243, 2004. ISSN 1388-3690. doi: http://dx.doi.org/10.1023/B:LISP.0000029445.29210.ca.

L. M. Kristensen and A. Valmari. Finding stubborn sets of coloured petri nets without unfolding. In *ICATPN '98: Proceedings of the 19th International Conference on Application and Theory of Petri Nets*, pages 104–123, London, UK, 1998. Springer-Verlag. ISBN 3-540-64677-9.

C. Lakos. The object orientation of object petri nets. In *Proceedings of the Workshop on Object Oriented Programming and Models of Concurrency*, pages 2–7, 1995.

G. T. Leavens and M. Sitaraman, editors. *Foundations of component-based systems*. Cambridge University Press, New York, NY, USA, 2000. ISBN 0-521-77164-1.

S. Leue and G. Holzmann. v-promela: A visual, object-oriented language for spin. In *ISORC '99: Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, page 14, Washington, DC, USA, 1999. IEEE Computer Society. ISBN 0-7695-0207-5.

A. G. Maguitman, F. Menczer, H. Roinestad, and A. Vespignani. Algorithmic detection of semantic similarity. In *Proceedings of the 14th international conference on World Wide Web*, WWW '05, pages 107–116, New York, NY, USA, 2005. ACM. ISBN 1-59593-046-9. URL http://doi.acm.org/10.1145/1060745.1060765.

T. Mailund and M. Westergaard. Obtaining memory-efficient reachability graph representations using the sweep-line method. In *Proceedings of the 10th International Conference Tools and Algorithms for the Construction and Analysis of Systems*, pages 177–191, 2004.

Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992. ISBN 0-387-97664-7.

S. McConnell. *Code Complete, Second Edition*. Microsoft Press, Redmond, WA, USA, 2004. ISBN 0735619670.

K. McMillan. *SMV Manual*, November 2000. URL http://www.cs.cmu.edu/~modelcheck/smv/smvmanual.ps.

S. Merz. Model checking: A tutorial overview. In *Modeling and Verification of Parallel Processes*, pages 3–38. Springer-Verlag, 2001.

A. Mukherjee. The structure of net 17. http://goanna.cs.rmit.edu.au/~amukherj/net17.pdf, December 2009a.

A. Mukherjee. Implementation of lookup algorithm. http://goanna.cs.rmit.edu.au/~amukherj/project.tar.gz, December 2009b.

A. Mukherjee, Z. Tari, and P. Bertok. Memory efficient state-space analysis in software model-checking. In *Thirty-Third Australasian Computer Science Conference (ACSC)*. CRPIT, 2010.

T. Murata. Petri nets: Properties, analysis and applications. In *Proceedings of the IEEE*, volume 77, pages 541–580, 1989.

G. J. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 1979. ISBN 0471043281.

C. Ouyang, E. Verbeek, W. van der Aalst, S. Breutel, M. Dumas, and A. ter Hofstede. WofBPEL: A tool for automated analysis of BPEL processes. In *Proceedings of the International conference on Service Oriented Computing - ICSOC 2005*, volume 3826, pages 484–489. Springer Berlin / Heidelberg, 2005.

C. Ouyang, E. Verbeek, W. M. P. van der Aalst, S. Breutel, M. Dumas, and A. H. M. ter Hofstede. Formal semantics and analysis of control flow in ws-BPEL. *Science of Computer Programming*, 67:162–198, July 2007. ISSN 0167-6423. doi: 10.1016/j.scico.2007.03.002. URL http://portal.acm.org/citation.cfm?id=1274201.1274469.

M. Paolucci, T. Kawamura, T. R. Payne, and K. P. Sycara. Semantic matching of web services capabilities. In *Proceedings of the First International Semantic Web Conference on The Semantic Web*, ISWC '02, pages 333–347, London, UK, 2002. Springer-Verlag. ISBN 3-540-43760-6. URL http://portal.acm.org/citation.cfm?id=646996.711287.

B. Parreaux. Difference compression in spin. In *Proceedings of the Fourth SPIN Workshop*, 1998.

S. R. Ponnekanti and A. Fox. Sword: A developer toolkit for web service composition. In *Proceedings of the 11th International World Wide Web Conference*, Honolulu, HI, USA, 2002.

A. Puntambekar. *Analysis and Design of Algorithms*. Technical Publications, first edition, 2008. ISBN 9788184313772.

A. V. Ratzer, L. Wells, H. M. Lassen, M. Laursen, J. F. Qvortrup, M. S. Stissing, M. Westergaard, S. Christensen, and K. Jensen. CPN tools for editing, simulating, and analysing coloured petri nets. In *Proceedings of the 24th International Conference on Applications and Theory of Petri Nets (ICATPN 2003), Eindhoven, The Netherlands, June 23-27, 2003 — Volume 2679 of Lecture Notes in Computer Science / Wil M. P. van der Aalst and Eike Best (Eds.)*, pages 450–462. Springer-Verlag, June 2003.

J. Rushby. Formal methods and critical systems in the real world. In *Formal Methods for Trustworthy Computer Systems (FM89)*, pages 121–125, 1989.

K. Schmidt. Using petri net invariants in state space construction. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003)*, pages 473–488. Springer Verlag, Feb. 2003.

K. Schmidt. LoLA: A low level analyser. In Nielsen, M. and Simpson, D., editors, *ICATPN 2000, Aarhus, Denmark, June 2000*, volume 1825, pages 465–474. Springer-Verlag, 2000.

K. Schmidt and C. Stahl. A petri net semantic for BPEL4WS validation and application. *Proceedings of the 11th Workshop on Algorithms and Tools for Petri Nets (AWPN 04)*, pages 1–6, 2004.

K. Schneider. *Verification of Reactive Systems: Formal Methods and Algorithms*. SpringerVerlag, 2004. ISBN 3540002960.

J. C. Sloan and T. M. Khoshgoftaar. From web service artifact to a readable and verifiable model. *IEEE Transacion on Services Computing*, 2:277–288, October 2009. ISSN 1939-1374. URL http://dx.doi.org/10.1109/TSC.2009.23.

C. Stahl. A petri net semantics for BPEL. Informatik-Berichte 188, Humboldt-Universitat zu Berlin, July 2005.

A. Tanenbaum. *Computer Networks*. Prentice Hall Professional Technical Reference, 2002. ISBN 0130661023.

S. Thatte. *XLANG Web Services for Business Process Design*. Microsoft Corporation, May 2001.

W. van der Aalst. Don't go with the flow: Web services composition standards exposed. *IEEE Intelligent Systems*, January/February 2003. doi: 10.1041/x1072s-2003. URL http://jmvidal.cse.sc.edu/library/aalst03a.pdf.

W. M. P. van der Aalst. The application of petri nets to workflow management. *Journal of Circuits, Systems, and Computers*, 8(1):21–66, 1998.

K. Varpaaniemi. Stable models for stubborn sets. *Fundamenta Informaticae*, 43:355–375, 2000.

C. F. Vasters. *Biztalk Server 2000: A Beginner's Guide*. McGraw-Hill Professional, 2001. ISBN 0072190116.

H. M. W. Verbeek, T. Basten, and W. M. P. van der Aalst. Diagnosing workflow processes using Woflan. *The Computer Journal*, 44(4):246–279, 2001. URL http://comjnl.oxfordjournals.org/cgi/reprint/44/4/246.

W. Visser. Memory efficient state storage in spin. In *Proceedings of the 2nd SPIN Workshop*, pages 21–35, 1996.

C. Walls and R. Breidenbach. *Spring in Action (Second Edition)*. Manning Publications Co., Greenwich, CT, USA, 2007. ISBN 1932394354.

M. Westergaard and L. Kristensen. Two interfaces to the cpn tools simulator. *Proceedings of Ninth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools (CPN'08)*, October 2008.

M. Westergaard, K. Jensen, S. Christensen, and L. Kristensen. *CPN Tools DTD*, December 2005. URL http://www.daimi.au.dk/~cpntools/bin/DTD/6/cpn.dtd.

P. Wohed, W. M. P. van der Aalst, M. Dumas, and A. H. M. ter Hofstede. Pattern-based analysis of BPEL4WS. *QUT Technical report, FIT-TR-2002-04*, 2002.

Y. Yang, Q. Tan, Y. Xiao, J. Yu, and F. Liu. Verifying web services composition: A transformation-based approach. In *Proceedings of the Sixth International Conference on Parallel and Distributed Computing Applications and Technologies*, pages 546–548, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2405-2.

X. Yi and K. J. Kochut. A cp-nets-based design and verification framework for web services composition. In *ICWS '04: Proceedings of the IEEE International Conference on Web Services*, page 756, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2167-3.

L. Zeng. *Dynamic web services composition*. PhD thesis, University of New South Wales, Australia, 2003. AAI0806727.